Assignment 6 Design Document

Zihua Li

CSE 13S: Computer Systems and C Programming

University of California, Santa Cruz

ABSTRACT

A design document of a program which identifies the top k likely authors that authored anonymous sample of text passed into stdin.

Keywords: Lempel-Ziv Compression, C Programming, LZ78

HASH TABLE

See ht.c

HashTable *ht_create(uint32_t size)

The constructor for a hash table. The size parameter denotes the number of slots that the hash table can index up to. The salt for the hash table is provided in salts.h.

```
Pseudocode
def ht_create(size):
    ht = {}
    ht[salt[0]] = SALT_HASHTABLE_LO
    ht[salt[1]] = SALT_HASHTABLE_HI
    ht[size] = size
    ht[insertions] = 0
    ht[slots]
    return ht
```

void ht$_{delete}(HashTable **ht)$

The destructor for a hash table. This should free any remaining nodes left in the hash table. Remember to set the pointer to the freed hash table to NULL.

```
Pseudocode
def ht_delete(ht):
    if (*ht)->slots AND (*ht)->size > 0:
        when i < (*ht)->size:
            if (*ht)->slots[i]:
                node_delete(&(*ht)->slots[i])
            i += 1
        (*ht)->size = 0
        (*ht)->slots = NULL
    *ht = NULL
```

uint32$_{t} ht_{s}ize(HashTable * ht)$

Returns the hash table's size, the number of slots it can index up to.

```
Pseudocode
def ht_size(ht):
    assert(ht)
    return ht->size
```

Node *ht$_{lookup}(HashTable*ht,char*word)$
Searches for an entry, a node, in the hash table that contains word. If the node is found, the pointer to the node is returned. Otherwise, a NULL pointer is returned.

```
Pseudocode
def ht_lookup(ht, word):
    assert(ht)
    if ht->insertion == 0:
        return NULL
    h = hash(ht->salt, word)
    hh = h % ht->size
    when i < ht->size:
        node = ht->slots[(hh + 1) % ht->size]
        if !node:
            return NULL
        if node-> == word:
            return node
    return NULL
```

Node *ht$_{insert}(HashTable*ht,char*word)$
Inserts the specified word into the hash table. If the word already exists in the hash table, increment its count by 1. Otherwise, insert a new node containing word with its count set to 1. Again, since we're using open addressing, it's possible that an insertion fails if the hash table is filled. To indicate this, return a pointer to the inserted node if the insertion was successful, and return NULL if unsuccessful.

```
Pseudocode
def ht_insert(ht, word):
    assert(ht)
    assert(word)
    assert(pointer of word)
    node = ht_lookup(ht, word)
    if !node:
        h = hash(ht->salt, word)
        hh = h % ht->size
        when i < ht->size:
            node = ht->slots[(hh + i) % ht->size]
            if !node:
                node = node_create(word)
                assert(node)
                ht->slots[(hh + i) % ht->size] = node
                break
        if !node:
            return NULL
        ht->insertions++
    node->count++
    return node
```

void ht$_{print}(HashTable*ht)$
A debug function to print out the contents of a hash table. Write this immediately after the constructor.

```
Pseudocode
def ht_print(ht):
    print(ht, ht->salt[0], ht->salt[1], ht->size, ht->insertions)
```

HashTableIterator *hti$_{c}reate(HashTable * ht)$
Creates a new hash table iterator. This iterator should iterate over the ht. The slot field of the iterator should be initialized to 0.

```
Pseudocode
def hti_create(ht):
    hti->table = ht
    hti->slot = 0
    hti->remaining = ht->insertions
    return hti
```

void hti$_{d}elete(HashTableIterator * *hti)$
Deletes a hash table iterator. You should not delete the table field of the iterator, as you may need to iterate over that hash table at a later time.

```
Pseudocode
def hti_delete(hti):
    (*hti)->table = NULL
    free(*hti)
```

Node *ht$_{i}ter(HashTableIterator * hti)$
Returns the pointer to the next valid entry in the hash table. This may require incrementing the slot field of the iterator multiple times to get to the next valid entry. Return NULL if the iterator has iterated over the entire hash table.

```
Pseudocode
def ht_iter(*hti):
    if hti->table->insertions:
        when hti->remaining > 0 AND hti->slot < hti->table->size:
            node = hti->table->slots[hti->slot]
            if node:
                hti->slot = (hti->slot + 1) % hti->table->size
                hti->remaining -= 1
                return node
            hti->slot = (hti->slot + 1) % hti->table->size
    return NULL
```

NODES

See node.c

Node *node$_{c}reate(char * word)$
The constructor for a node. You will want to make a copy of the word that is passed in. This will require allocating memory and copying over the characters for the word. You may find strdup() useful.

```
Pseudocode
def node_create(word):
    assert(n)
    n->word = strdup(word)
    n->count = 0
    return n
```

void node$_{d}elete(Node * *n)$
The destructor for a node. Since you have allocated memory for word, remember to free the memory allocated to that as well. The pointer to the node should be set to NULL.

```
Pseudocode
def node_delete(**n):
    free((*n)->word)
    (*n)->word = NULL
    free(*n)
    *n = NULL
```

void node$_{print}(Node * n)$
A debug function to print out the contents of a node.

```
Pseudocode
def node_print(Node *n):
    print(n, n->word, n->count)
```

BLOOM FILTERS

See bf.c

BloomFilter *bf$_{c}reate(uint32_{t}size)$
The constructor for a Bloom filter. The primary, secondary, and tertiary salts that should be used are provided in salts.h. Note that you will also have to implement the bit vector ADT for your Bloom filter, as it will serve as the array of bits necessary for a proper Bloom filter. Bit vectors will be discussed in §8.

```
Pseudocode
def bf_create(size):
    bf->primary[0] = SALT_PRIMARY_LO
    bf->primary[1] = SALT_PRIMARY_HI
    bf->secondary[0] = SALT_SECONDARY_LO
    bf->secondary[1] = SALT_SECONDARY_HI
    bf->tertiary[0] = SALT_TERTIARY_LO
    bf->tertiary[1] = SALT_TERTIARY_HI
    bf->filter = bv_create(size)
    return bf
```

void bf$_{d}elete(BloomFilter * *bf)$
The destructor for a Bloom filter. As with all other destructors, it should free any memory allocated by the constructor and null out the pointer that was passed in.

```
Pseudocode
def bf_delete(**bf):
    assert(bf)
    assert(*bf)
    bv_delete(&(*bf)->filter)
    (*bf)->filter = NULL
    free(*bf)
    *bf = NULL
```

uint32$_{t}bf_{s}ize(BloomFilter * bf)$
Returns the size of the Bloom filter. In other words, the number of bits that the Bloom filter can access. Hint: this is the length of the underlying bit vector.

```
Pseudocode
def bf_delete(**bf):
    assert(bf)
    assert(bf->filter)
    return bv_length(bf->filter)
```

void bf$_{i}nsert(BloomFilter * bf, char * oldspeak)$
Takes word and inserts it into the Bloom filter. This entails hashing word with each of the three salts for three indices, and setting the bits at those indices in the underlying bit vector.

```
Pseudocode
def bf_insert(bf, oldspeak):
     assert(bf)
     assert(bf->filter)
     assert(oldspeak)
     bfSize = bf_size(bf)
     h = hash(bf->primary, oldspeak)
     bv_set_bit(bf->filter, h % bfSize)
     h = hash(bf->secondary, oldspeak)
     bv_set_bit(bf->filter, h % bfSize)
     h = hash(bf->tertiary, oldspeak)
     bv_set_bit(bf->filter, h % bfSize)
```

bool bf$_{p}robe(BloomFilter * bf, char * oldspeak)$
Probes the Bloom filter for word. Like with bf_insert(), word is hashed with each of the three salts for three indices. If all the bits at those indices are set, return true to signify that word was most likely added to the Bloom filter. Else, return false.

```
Pseudocode
def bf_delete(**bf):
     assert(bf)
     assert(bf->filter)
     assert(oldspeak)
     bfSize = bf_size(bf)
     h = hash(bf->primary, oldspeak)
     if !bv_get_bit(bf->filter, h % bfSize)
          return false
     h = hash(bf->secondary, oldspeak)
     if !bv_get_bit(bf->filter, h % bfSize)
          return false
     h = hash(bf->tertiary, oldspeak)
     return bv_get_bit(bf->filter, h % bfSize)
```

void bf$_{p}rint(BloomFilter * bf)$
A debug function to print out the bits of a Bloom filter. This will ideally utilize the debug print function you implement for your bit vector.

```
Pseudocode
def bf_delete(**bf):
     print(bf, bf->primary[0], bf->primary[1], bf->secondary[0], bf->secondary[1
```

BIT VECTORS

See bv.c

BitVector *bv$_{c}reate(uint32_{t}length)$

The constructor for a bit vector that holds length bits. In the even that sufficient memory cannot be allocated, the function must return NULL. Else, it must return a BitVector *, or a pointer to an allocated BitVector. Each bit of the bit vector should be initialized to 0.

```
Pseudocode
def bv_create(length):
     assert(bv)
```

```
    bv->length = length
    byteLength = (length / 8) + 1
    assert(bv->vector)
    return bv
```

void bv$_{d}elete(BitVector**bv)$
The destructor for a bit vector. Remember to set the pointer to NULL after the memory associated with the bit vector is freed.

```
Pseudocode
def bv_delete(**bv):
    assert(bv)
    assert(*bv)
    free((*bv)->vector)
    (*bv)->vector = NULL
    free(*bv)
    *bv = NULL
```

uint32$_{t}bv_{l}ength(BitVector*bv)$
Returns the length of a bit vector.

```
Pseudocode
def bv_length(*bv):
    assert(bv)
    return bv->length
```

bool bv$_{s}et_{b}it(BitVector*bv, uint32_{t}i)$
Sets the ith bit in a bit vector. If i is out of range, return false. Otherwise, return true to indicate success.

```
Pseudocode
def bv_set_bit(*bv, i):
    assert(bv)
    assert(bv->vector)
    if i < bv->length:
        byteIndex = i / 8
        bitOffset = i % 8
        bv->vector[byteIndex] |= 1 << bitOffset
        return true
    return false
```

bool bv$_{c}lr_{b}it(BitVector*bv, uint32_{t}i)$
Clears the i th bit in the bit vector. If i is out of range, return false. Otherwise, return true to indicate success.

```
Pseudocode
def bv_clr_bit(*bv, i):
    assert(bv)
    if i < bv->length:
        byteIndex = i / 8
        bitOffset = i % 8
        bv->vector[byteIndex] &= ~(1 << bitOffset)
        return true
    return false
```

bool bv$_{get_b}it(BitVector*bv, uint32_t i)$
Returns the i th bit in the bit vector. If i is out of range, return false. Otherwise, return false if the value of bit i is 0 and return true if the value of bit i is 1.

```
Pseudocode
def bv_get_bit(*bv, i):
    assert(bv)
    if i < bv->length:
        byteIndex = i / 8
        bitOffset = i % 8
        return (bv->vector[byteIndex] & ~(1 << bitOffset)) != 0
    return false
```

void bv$_{p}rint(BitVector*bv)$
A debug function to print the bits of a bit vector. That is, iterate over each of the bits of the bit vector. Print out either 0 or 1 depending on whether each bit is set. You should write this immediately after the constructor.

```
Pseudocode
def bv_print(*bv):
    i = 1
    when i < bv->length:
        if i * 8 == 0:
            print(",")
        print(bv_get_bit(bv, i))
        i += 1
```

TEXTS

See text.c

Text *text$_{c}reate(FILE*infile, Text*noise)$
The constructor for a text. Using the regex-parsing module, get each word of infile and convert it to lowercase. The noise parameter is a Text that contains noise words to filter out. That is, each parsed, lowercase word is only added to the created Text if and only if the word doesn't appear in the noise Text.

```
Pseudocode
def text_create(infile, *noise):
    assert(infile)
    regex_t wordPattern
    if regcomp(&wordPattern, "[A-Za-z]+([-'][A-Za-z]+)*", REG_EXTENDED):
        fprintf(stderr, "Failed to compile regex.\n")
        return NULL
    text->ht = ht_create(1 << 19)
    assert(text->ht)
    text->bf = bf_create(1 << 21)
    assert(text->bf)
    text->word_count = 0
    char *word = NULL
    while ((noise || text->word_count < optNumberOfNoiseWordsToFilterOut) \
    && ((word = next_word(infile, &wordPattern)) != NULL)):
        if !noise || !text_contains(noise, word):
            ht_insert(text->ht, word)
            bf_insert(text->bf, word)
            text->word_count++
        free(word)
    fclose(infile)
    return text
```

void text$_{delete}(Text **text)$

Deletes a text. Remember to free both the hash table and the Bloom filter in the text before freeing the text itself. Remember to set the pointer to NULL after the memory associated with the text is freed.

```
Pseudocode
def text_delete(**text):
    assert(text)
    assert(*text)
    ht_delete(&(*text)->ht)
    bf_delete(&(*text)->bf)
    free(*text)
    *text = NULL
```

double text$_{dist}(Text *text1, Text *text2, Metric metric)$

This function returns the distance between the two texts depending on the metric being used. This can be either the Euclidean distance, the Manhattan distance, or the cosine distance. The Metric enumeration is provided to you in metric.h and will be mentioned as well in §12.

```
Pseudocode
Too long, no time to provide
```

double text$_{frequency}(Text *text, char *word)$

Returns the frequency of the word in the text. If the word is not in the text, then this must return 0. Otherwise, this must return the normalized frequency of the word.

```
Pseudocode
def text_frequency(*text, *word):
    assert(text)
    assert(text->ht)
    assert(word)
    Node *node = ht_lookup(text->ht, word)
    if node:
        return (double)node->count / (double) text->word_count
    return 0.0
```

bool text$_{contains}(Text *text, char *word)$

Returns whether or not a word is in the text. This should return true if word is in the text and false otherwise.

```
Pseudocode
def text_contains(Text *text, char *word) {
    assert(text)
    assert(text->ht)
    assert(word)
    if bf_probe(text->bf, word):
        return ht_lookup(text->ht, word) != NULL
    return NULL
```

void text$_{print}(Text *text)$

A debug function to print the contents of a text. You may want to just call the respective functions of the component parts of the text.

```
Pseudocode
def text_contains(Text *text, char *word) {
    assert(text)
    assert(text->ht)
    assert(word)
    if bf_probe(text->bf, word):
        return ht_lookup(text->ht, word) != NULL
    return NULL
```

ENCODE.C

Contains the main() function for the encode program.

main(int argc, char *const argv[])

This is the main function.

```
Pseudocode
def main(argc, argv[]):
    user_input = input("menu_message")
    if user_input == a:
        optAscii++
    elif user_input == i:
        inPath = strdup(option_argument)
        # assume it was split from user_input
    elif user_input == o:
        outPath = strdup(option_argument)
        break
    elif user_input == v:
        optVerbose++
        break
    elif user_input == h:
        print("menu_message")
        return
    else:
        print("menu_message")
        return
    #open infile
    inFile = open(inPath)
    #file header
    FileHeader outFileHeader = {0xBAADBAAC, inFileStats.st_mode}
    #open outfile
    outFile = open(outPath)
    #write header
    write_header(outFile, &outFileHeader)
    #create trie
    trieRoot = trie_create()
    curr_node = trieRoot
    #a monotonic counter to keep track of the next available code
    next_code = START_CODE
    #two variables to keep track of the previous trie node
    #and previously read symbol
    prev_node = NULL
    prev_sym = NULL
    #read all symbols from infile
    curr_sym = NULL
    while read_sym(inFile, &curr_sym):
        next_node = trie_step(curr_node, curr_sym)
        if next_node:
            prev_node = curr_node
            curr_node = next_node
        else:
            write_pair(outFile, curr_node->code, curr_sym, /
                bitLength(next_code))
            curr_node->children[curr_sym] = \
                trie_node_create(next_code)
            curr_node = trieRoot
            next_code = (next_code + 1) % (MAX_CODE + 1)
        if next_code == MAX_CODE:
            next_code = START_CODE
            trie_reset(trieRoot)
        prev_sym = curr_sym
        if curr_node != trieRoot:
            write_pair(outFile, prev_node->code, prev_sym, \
                bitLength(next_code))
            next_code = (next_code + 1) % (MAX_CODE + 1)
        #write the pairr (STOP_CODE, 0) to signal the end
        write_pair(outFile, STOP_CODE, 0, bitLength(next_code))
        #flush pairs
        flush_pairs(outFile)
        #close both files
        close(inFile)
        close(outFile)
        return
```

DECODE.C

Contains the main() function for the decode program.

main(int argc, char *const argv[])
This is the main function.

```
Pseudocode
def main(argc, argv[]):
    user_input = input("menu_message")
    if user_input == a:
        optAscii++
    elif user_input == i:
        inPath = strdup(option_argument)
        # assume it was split from user_input
    elif user_input == o:
        outPath = strdup(option_argument)
        break
    elif user_input == v:
        optVerbose++
        break
    elif user_input == h:
        print("menu_message")
        return
    else:
        print("menu_message")
        return
    #open infile
```

```
inFile = open(inPath)
#file header
FileHeader outFileHeader = {0xBAADBAAC, inFileStats.st_mode}
#create WordTable
wordTable = wt_create();
#write header
write_header(outFile, &outFileHeader)
#keep track of the current code and next code
curr_node = NULL
next_code = START_CODE
#read all pairs from infile
curr_sym = NULL
while read_pair(inFile, &curr_code, &curr_sym, \
    bitLength(next_code)):
    wordTable[next_code] = \
        word_append_sym(wordTable[curr_code], curr_sym)
    write_word(outfile, wordTable[next_code])
    if (++next_code == MAX_CODE):
        next_code = START_CODE
        wt_reset(wordTable)
#flush
flush_words(outFile)
#close both files
close(inFile)
close(outFile)
```

WORKS CITED:

Assignment document provided by Professor.