Karthi Sankar

kasankar@ucsc.edu

11/28/21

CSE 13S Assignment 7 Design Document

Purpose:

The purpose of this assignment is to filter out the content given as input to the program. Certain words will be classified depending on what kind of word it is. From there, the "user" of the language will be either punished, reprimanded, or corrected. This assignment is essentially the parsing of input with the use of multiple different ADTs that will be implemented. The words "spoken" by the user and how "bad" they are will be the basis of how they are dealt with and what output the program has.

Structure:

The program has a main file called banhammer.c that will be used for the handling of input and the output of the program. The ADTs that will be implemented and used are ht.c (hash table), bst.c (binary search tree), node.c, bf.c (bloom filter), bv.c (bit vector). Another file called parser.c will be used to parse through the input of the file with the use of regular expressions. The ADTs' main use will be to check if the words used by the user are acceptable, bad, or inexcusable. The words will be classified using hashing and bloom filters. The bloom filter will be used to see if a word has already been seen or classified.

(further explanation in final design)

Files (pseudocode + description):

bv.c:

```
BitVector *bv_create(uint32_t length):
    set length to parameter length
    dynamically allocate the vector array to be of size length
    return pointer to bit vector

void bv_delete(BitVector **bv):
    if the vector array exists:
        free the vector array
        vector = NULL
    return

uint32_t bv_length(BitVector *bv):
    return bv.length

bool bv_set_bit(BitVector *bv, uint32_t i):
    set vector[i/8] to 1 with vector[i] |= (0x1 << i % 8)

bool bv_clr_bit(BitVector *bv, uint32_t i):
    clear vector[i/8] with vector[i] &= ~(0x1 << i % 8)

bool bv_get_bit(BitVector *bv, uint32_t i):
    return (vector[i/8] >> i % 8) & 0x1   # get bit at index i

void bv_print(BitVector *bv):
    print the vector array
```

The function bv_create() returns a pointer to a bit vector and takes in a 32-bit unsigned integer called length. First, the length of the vector is set, and the vector array is dynamically allocated and will be the size of the length variable. Finally, the bit vector is returned. Bv_delete() returns void and takes in a double pointer to a bit vector. It is used to free memory used by the bit vector, namely the vector array. The function bv_length() returns the 32-bit unsigned integer variable length by taking in a bit vector as a parameter.

The bit vector will have bits set, cleared, and “gotten”. This will be done using bitwise operations. The vector array will use [n/8] unsigned 8-bit integers in order to access 8 indices with a single integer access in order to make this process efficient. The function bv_set_bit() will be used to set a bit, returning a boolean signaling success and taking in a bit vector and an index as a parameter. If the index being set is out of range, false is returned. Because a bit is an 8-bit integer, the index to be accessed is i/8. Then, to place a one in the vector array, a bitwise or is used. Essentially, a 1 is left-shifted by (i % 8) places in an 8-bit number (because of a bit's size). Then that number is bitwise-ored with the index [i/8] in the array vector. Because a 0 and 1 orred together will always yield a 1, this results in a 1 being placing the index necessary within the array vector. Then, a true is returned, signaling a successful setting of a bit.

The function bv_clr_bit() will be used to clear a bit, returning a boolean signaling success and taking in a bit vector and an index as a parameter. If the index being cleared is out of range, false is returned. Because a bit is an 8-bit integer, the index to be accessed is i/8. Then, to place a one in the vector array, a bitwise and is used. Essentially, a 1 is left-shifted by (i % 8) places in an 8-bit number (because of a bit's size). Then that number is bitwise-anded with the index [i/8] in the array vector. Because a 0 and 1 anded together will always yield a 0, this results in a 0 being placing the index necessary within the array vector. Then, a true is returned, signaling a successful clearing of a bit.

The function bv_get_bit() gets a bit, returning a boolean signaling what value is at an index and taking in a bit vector and an index as a parameter. This is done using bit operations, including bitwise and and the right shift operator. Because a bit is an 8-bit integer, the index to be accessed is i/8. In order to have the same value at a certain index be returned, a bitwise and is used. This is because a value and 1 will always return the value of the number given. Then, the whole operation of getting a bit is returned, because this will signal that the getting of the bit was successful. For 0, false is returned; for 1, true is returned.

The final function, bv_print() prints a bit vector that is passed as a parameter to the function.

bf.c:

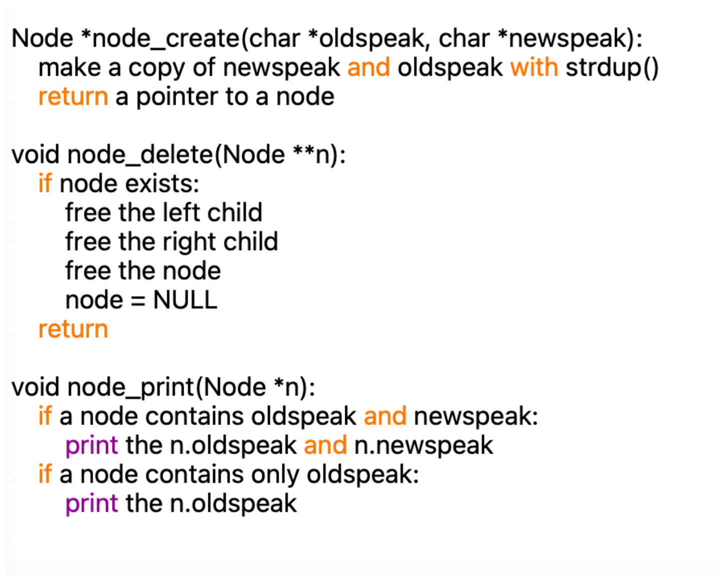
```
static int count = 0;

BloomFilter *bf_create(uint32_t size):
    set elements in primary salts array
    set elements in secondary salts array
    set elements in tertiary salts array
    dynamically allocate the filter array to be the size, size
    return pointer to bloom filter

void bf_delete(BloomFilter **bf):
    if filter exists:
        free the filter array
        filter = NULL
    return

uint32_t bf_size(BloomFilter *bf):
    return bf.size

void bf_insert(BloomFilter *bf, char *oldspeak):
    hash with primary salt and oldspeak
    hash with secondary salt and oldspeak
    hash with tertiary salt and oldspeak
    set resulting indices in filter array
    count += 1

bool bf_probe(BloomFilter *bf, char *oldspeak):
    hash with primary salt and oldspeak
    hash with secondary salt and oldspeak
    hash with tertiary salt and oldspeak
    if all three resulting indices are set:
        return true
    #end if
    return false

uint32_t bf_count(BloomFilter *bf):
    return count

void bf_print(BloomFilter *bf):
    print filter array
```

First, I initialize a static variable count to count the number of bits that have been set in a bloom filter. The function bv_create() is responsible for creating a bloom filter, returning a pointer to it. First, the salts arrays are set to the necessary values; then, the bloom filter array is dynamically allocated to be the size of the parameter size, and the bloom filter is returned. Bv_delete() will free all memory associated with the bloom filter, specifically the filter array. The function bv_size() returns the size of the bloom filter, a 32-bit unsigned integer. The function bv_count() returns the number of bits that are set in the bloom filter, a 32-bit unsigned integer.

The function bf_insert() has a return type of void and takes in a bloom filter and oldspeak, a pointer to characters, inserting oldspeak into a bloom filter. First, oldspeak is hashed using each of the three different salts. Then, each index that results from the hash function calls are set in the bloom filter using bv_set_bit(). Finally, I increment the count of bits that have been set. The function bf_probe() has a return type of void and takes in a bloom filter and oldspeak, a pointer to characters, probing the filter for oldspeak. First, oldspeak is hashed using each of the three different salts. Then, each index that results from the hash function calls are checked in the bloom filter to see if they are set (using bf_get_bit()); if they are all set, true is returned. If they are not set, false is returned.

The final function, bf_print() prints a bloom filter that is passed as a parameter to the function.

node.c:



```
Node *node_create(char *oldspeak, char *newspeak):
    make a copy of newspeak and oldspeak with strdup()
    return a pointer to a node

void node_delete(Node **n):
    if node exists:
        free the left child
        free the right child
        free the node
        node = NULL
    return

void node_print(Node *n):
    if a node contains oldspeak and newspeak:
        print the n.oldspeak and n.newspeak
    if a node contains only oldspeak:
        print the n.oldspeak
```

The function node_create() returns a pointer to a node and takes in the character pointers oldspeak and newspeak as parameters. First, copies of oldspeak and newspeak are saved using the function strdup(). Then, the pointer to the node is returned.

(My current pseudocode for node_delete() is not accurate)

The function node_delete() takes in a double pointer to a node and frees all memory associated with a node. Its return type is void. If oldspeak and newspeak are not NULL, they are freed. Then if the node exists, the node n is freed and set to NULL.

The function node_print() prints the contents of a node, depending on if it contains oldspeak or newspeak.

bst.c:

(not fully implemented, will be in final design doc)

ht.c:

(not fully implemented, will be in final design doc)

parser.c:

(not fully implemented, will be in final design doc)

banhammer.c:

(not fully implemented, will be in final design doc)