Nayeel Imtiaz
11/28/21
CSE 13s

Assignment 7 Design Document

Summary: This project is all about bloom filters, binary trees, and hash tables. A bloom filter is a space efficient probability based data structure. A bloom filter can be represented by an array of bit vectors. A good hash function is needed for the bloom filter to work. Encryption is the process of taking some file you wish to protect and making sure only authorized people can access it. A bit vector is an abstract data type that represents a one dimensional array of bits. A hash table is a data structure that maps keys to values for fast access. A node file is also needed to complete this project. A binary search tree is a data structure that is basically a regular binary tree but sorted.

Pseudocode:

Bf.c:

```
BloomFilter *bf_create(uint32_t size):
        Dynamically allocate memory to create a bloom filter.
        primary[0] = SALT_PRIMARY_LO
        primary[1] = SALT_PRIMARY_HI
        secondary[0] = SALT_SECONDARY_LO
        secondary[1] = SALT_SECONDARY_HI
        tertiary[0] = SALT_TERTIARY_LO
        tertiary[1] = SALT_TERTIARY_HI
        filter = bv_create(size)

void bf_delete(BloomFilter **bf):
        Delete the memory allocated for the bloom filter.
        bv_delete(bf)
        Bf = NULL

uint32_t bf_size(BloomFilter *bf):
        Return the size of the bloom filter.

void bf_insert(BloomFilter *bf, char *oldspeak):
    bv_set_bit(filter, hash(primary, oldspeak) % bf_size(bf))
    bv_set_bit(filter, hash(secondary, oldspeak) % bf_size(bf))
    bv_set_bit(filter, hash(tertiary, oldspeak) % bf_size(bf))

bf_probe(BloomFilter *bf, char *oldspeak)
    Return bv_set_bit(filter, hash(primary, oldspeak) % bf_size(bf)) &
```

bv_set_bit(filter, hash(secondary, oldspeak) % bf_size(bf)) &
bv_set_bit(filter, hash(tertiary, oldspeak) % bf_size(bf))

uint32_t bf_count(BloomFilter *bf):
    Count = 0
    For i in range(size of bf)
        If bf[i] not null
            Count += 1
    Return count

void bf_print(BloomFilter *bf):
    Print the bloom filter using printf function.

Bv.c:

BitVector *bv_create(uint32_t length):
    Dynamically allocate memory for a bit vector.
    bv_length = length
    bv_vector_memory = ceil(length / 8.0) * (memory of uint8_t variable)

void bv_delete(BitVector **bv):
    Delete the memory allocated for the bit vector.
    Free(bv)
    Bv = NULL

uint32_t bv_length(BitVector *bv):
    Return the length of the bit vector.

void bv_print(BitVector *bv):
    Print the bit vector.

Ht.c:

HashTable *ht_create(uint32_t size):
    Dynamically allocate memory for a hash table.
    Ht_size = size
    Salt[0] = low salt value
    Salt[1] = high salt value
    Ht_tree = create(node)

```
void ht_delete(HashTable **ht):
            for i in range(hash table size)
                  Delete tree
          free(ht)
          Ht = NULL

ht_lookup(HashTable *ht, char *oldspeak)
   lookups += 1
   index = hash(salt, oldspeak) % size
   return bst_find(trees[index], oldspeak)


ht_insert(HashTable *ht, char *oldspeak, char *newspeak)
   lookups += 1
   uint32_t index = hash(salt, oldspeak) % size
   trees[index] = bst_insert(trees[index], oldspeak, newspeak)

ht_count(HashTable *ht)
   count = 0
   for i in range(hash table size)
      if trees[i] != NULL
         count += 1
   return count



ht_size(HashTable *ht)
          Return the size of the hash table.

ht_print(HashTable *ht):
          Print the hash table.

ht_avg_bst_size(HashTable *ht):
   Average size = 0
   for i in range(hash table size)
      Average size += bst_size(trees[i])

   return  (Average size) / ht_count(ht)
```

```
double ht_avg_bst_height(HashTable *ht):
    Average height = 0
    for i in range(hash table size)
        Average height += bst_height(trees[i])

    return  (Average height) / ht_count(ht)
```

Node.c:

```
Node node_create():
        Create a node by dynamically allocating space for it.
        Oldspeak = NULL or string1
        Newspeak = NULL or string2
        Left = NULL
        Right = NULL

void node_delete(Node **n):
        Free oldspeak
        Free newspeak
        Free n

void node_print(Node *n):
        Print the node using printf functions.
```

Bst.c:

```
bst_create(void)
    return NULL

void bst_delete(Node **root)
    If root != NULL
        bst_delete(root_left)
        bst_delete(root_right)
        node_delete(root)

uint32_t bst_height(Node *root)
    if root = true:
        return max(bst_height(root_left), bst_height(root_right)) + 1
```

```
bst_size(Node *root):
    if root == NULL
          return 0
    return bst_size(root_left) + bst_size(root_right) + 1


bst_find(Node *root, char *oldspeak):
    if root != NULL && oldspeak != NULL
        while curr != NULL && curr_oldspeak != oldspeak
            branches += 1
            if curr_oldspeak > oldspeak
                curr = curr_left

            else
                curr = curr_right
    return curr

bst_insert(Node *root, char *oldspeak, char *newspeak):
    if root == NULL:
        return node_create(oldspeak, newspeak

    if root_oldspeak > oldspeak:
        root_left = bst_insert(root_left, oldspeak, newspeak)
        branches += 1

    else if root_oldspeak < oldspeak:
        root_right = bst_insert(root_right, oldspeak, newspeak)
        branches += 1

 return root
```

Banhammer.c

```
  opt = 0
  no_input = true
  h_flag = false
  s_flag = false
  t_flag = false
  f_flag = false
```

ht_size_set = 2^16
bf_size_set = 2^20

Parse through command line arguments using get opt.
If h:
    Show help commands
If s:
    Turn on statistics
If t:
    Change default hash table size
If f:
    Change default bloom filter size

Create bloom filter and hash table
Scan bad file and put contents into badspeak buffer.
Scan new file and put contents into oldspeak and newspeak buffers.

Create two binary search trees called badmsg and mixedmsg.

While words are left to be read:
    For i in range(size of word):
        Make letter lowercase

If stats enabled:
    Print stats

If mixed words > 0 and bad words == 0:
    Print good message
If mixed words > 0 and bad words > 0:
    Print mixed message
If mixed words == 0 and bad words > 0:
    Print bad message

Free all memory that was allocated.