Karthi Sankar

kasankar@ucsc.edu

12/05/21

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 and hash table will be used to see if a word has already been seen or classified.

Files (pseudocode + description):

bv.c:

CITE: Functions bv_set_bit(), bv_clr_bit(), bv_get_bit() are inspired from Professor Long's file bv8.c in the Code Comments Repo

```
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):
   Node *n = malloc() pointer to a node
   if (oldspeak != NULL):
      n.oldspeak = NULL
   else:
      n.oldspeak = strdup(oldspeak)
   if (newspeak != NULL):
      n.newspeak = NULL
   else:
      n.newspeak = strdup(newspeak)
   n.left and right = NULL
   return n

void node_delete(Node **n):
   if node exists:
      free n.oldspeak
      free n.newspeak
      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, I allocate a node pointer using malloc(). Then, depending on if the oldspeak and newspeak parameters are NULL or not, (if they are not NULL) I make copies of oldspeak and newspeak (for the node itself) using the function strdup(). Then, I set the node's left and right children to be NULL. Then, the pointer to the node is returned.

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. Then if the node exists, then first, its oldspeak and newspeak are freed. Then, the node pointer 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:

CITE: Some Functions are inspired/come from Professor Long's Lecture 18 about Trees

max(): Lecture 18 slide 55

bst_height(): Lecture 18 slide 55

bst_size(): inspired from bst_height()

bst_find(): Lecture 18 slide 57

bst_insert(): Lecture 18 slide 62

bst_print(): Lecture 18 slides 22-34

bst_delete(): Lecture 18 slides 79-85

This binary search tree that we are implementing is used to store the "flagged" words that are not good. It is ordered, and the words will be stored in lexicographical order. The tree is made up of

Node pointers, and the number of branches traversed will be tracked via an extern variable named branches.

```
int branches = 0

Node *bst_create(void):
    return NULL

void bst_delete(Node **root):
    if root exists:
        bst_delete(root.left)
        bst_delete(root.right)
        node_delete(root)
```

The function bst_create returns a Node pointer and has no parameters. The function will be used to return an empty binary search tree, which consists of a NULL pointer, so all the function contains is a return of NULL. bst_delete is responsible for the deletion of and freeing of any memory that is taken by a binary search tree. It has no return type and takes in a double pointer to the root node of the tree. A postorder traversal is used to free all the memory. So, if the root is not NULL, then bst_delete() is called on the root's left and right children. Finally, node_delete() is called on the root. Through this traversal, all nodes in the tree will be deleted.

```
static int max(int x, int y):
    return x > y ? x : y

int bst_height(Node *root):
    if root exists:
        return 1 + bst_height(root.left) + bst_height(root.right)
    #end if
    return 0

int bst_size(Node *root):
    if root == NULL:
        return 0
    if root exists:
        return 1 + bst_size(root.left) + bst_size(root.right)
    #end if
```

The function bst_height() makes use of a static function named max() that returns the maximum number between two integers. Bst_height() returns a 32-bit unsigned integer and takes in a node pointer to the root of a tree. If the root exists, it must have a size of 1. To 1, I add the height of the tallest two subtrees (left and right children) (using a call to bst_height()). If there is no root, I return 0 as the height.

The function bst_size() is very similar. Bst_size() returns a 32-bit unsigned integer and takes in a node pointer to the root of a tree. If the root exists, it must have a size of 1. To 1, I add the size of the tallest two subtrees (left and right children) (using a call to bst_size()). If there is no root, I return 0 as the height.

```
Node *bst_find(Node *root, char *oldspeak):
   if root == NULL:
      return NULL
   if root exists:
      if root.oldspeak > oldspeak:
         branches += 1
         return bst_find(root.left, oldspeak)
      if root.oldspeak < oldspeak:
         branches += 1
         return bst_find(root.right, oldspeak)
      #end if
   #end if
   return root

Node *bst_insert(Node *root, char *oldspeak, char *newspeak):
   if root exists and oldspeak != NULL:
      if root.oldspeak > oldspeak:
         branches += 1
         root.left = bst_insert(root.left, oldspeak, newspeak)
      else:
         branches += 1
         root.right = bst_insert(root.right, oldspeak, newspeak)
      #end if
      return root
   #end if
   return node_create(oldspeak, newspeak)

void bst_print(Node *root):
   if root exists:
      bst_print(root.left)
      node_print(root)
      bst_print(root.right)
   #end if
```

The function bst_find is used to find a node in the binary search tree. The function returns a Node pointer and takes a root node pointer and the oldspeak of the node to be found. The initial check within the function is to check if the root node is null; if it is, the tree is empty, so NULL is returned. If the root isn't null, the tree is searched recursively. If the root's oldspeak is lexicographically greater than the oldspeak to be found, the left side of the tree is searched with a call to bst_find() of the root's left node; if the root's oldspeak is lexicographically less than the oldspeak to be found, the right side of the tree is searched with a call to bst_find() of the root's right node. In each of these cases, the number of branches is incremented because more branches are traversed with each recursive call. If neither of these two are the case, then the node to be found is the root node itself, so the root node is returned.

The function bst_insert() is used to insert a node into the binary search tree. The function returns a Node pointer and takes the root node pointer and the oldspeak and the newspeak of the node to be inserted. The function returns the root of the tree in which the node is inserted, and the root will be updated when a node is inserted if necessary. First, there is a check if the tree is empty (root is NULL) or if the oldspeak parameter is NULL, then immediately a node is returned with the specified oldspeak and newspeak because that is the first node in the tree. If the tree isn't empty, the structure is exactly the same as bst_find(). If the root's oldspeak is lexicographically greater than the oldspeak (of the node to be inserted), the left side of the tree will be checked for insertion with a call to bst_insert() of the root's left node; if the root's oldspeak is lexicographically less than the oldspeak (of the node to be inserted), the right side of the tree will be checked for insertion a call to bst_insert() of the root's right node. In each of these cases, the number of branches is incremented because more branches are traversed with each recursive call.

If neither of these two are the case, then the node to be inserted is a duplicate, so nothing is inserted and the same root node is returned.

The function bst_print() has a void return type and takes in a double pointer to a root node. This function is used to print the binary search tree. An inorder traversal is used to print out the nodes of the tree because all of the nodes will be printed in lexicographical (and alphabetical) order. So if the root of the tree isn't NULL, the three calls in the function are as follows: a recursive bst_print() call to the root's left child, a call to node_print() of the root node, and finally a recursive bst_print() call to the root's right child. As a result, all the nodes of the BST will be printed in order.

ht.c:

CITE: Eugene explained the functions ht_insert() and ht_lookup() in his 11/30 tutoring section

A hash table is a data structure that maps keys to values and has O(1) (very fast) lookup times. We use in our case to store words by hashing the word to be inserted using a hach function and placing it at the appropriate index. The hash table contains binary search trees in order to prevent hash collisions and duplicate words will never be inserted into a tree. The number of lookups to the hash table will be tracked using an extern variable. The hash table itself is an array of binary search trees (containing node pointers). The hash table has salts and a set size.

```
HashTable *ht_create(uint32_t size):
   HTpointer ht = malloc() a HT pointer
   if ht exists:
      set salts using salts array (from salts.h)
      ht.size = size
      ht.trees = calloc array of Node pointers
      for i in range (0, size):
         ht.trees[i] = bst_create()
      #end for
   #end if
   return ht

void ht_delete(HashTable **ht):
   for i in range (0, ht.size):
      bst_delete(ht.trees[i])
   #end for
   free(ht.trees)
   free(ht)
   ht = NULL
```

The function ht_create() returns a HashTable pointer and takes in a parameter as for its size. It returns a created hash table. First, the HashTable pointer ht is allocated using malloc(). Then, if the malloc was successful, the salts are set (from the header file) and the size of set. Then, the array of BSTs (called trees) is dynamically allocated using calloc() to be full of node pointers. Then, each index of the array trees is set to an empty BST (using the function bst_create()). Lastly, the HashTable pointer is returned. The function ht_delete has a void return type and takes a double pointer to a hash table. First, the array trees is cleared using a for loop; for each index in the array, the tree at index is deleted with bst_delete(). Then, the trees array is freed, and then the HashTable pointer is freed and set to NULL.

```
int ht_size(HashTable *ht):
   return ht.size

int ht_count(HashTable *ht):
   int count = 0
   for i in range(0, ht_size(ht)):
      if ht.trees[i] != NULL:
         count += 1
      #end if
   #end for
   return count

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

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak):
   lookups += 1
   int index = hash(ht.salt, oldspeak) % ht.size
   ht.trees[index] = bst_insert(ht.trees[index], oldspeak, newspeak)
```

The function ht_size returns a 32-bit unsigned integer and returns the size of the hash table that is passed into the function. The function ht_count() returns the number of non-NULL BSTs in the hash table. It returns a 32-bit unsigned integer, and takes in a HashTable pointer. A variable count is initialized to 0. Then, I iterate through the trees array, and if the BST at index i is not equal to NULL, I increment count by 1. Finally, I return the count.

The function ht_lookup() is used to find a node in the hash table that contains the specified oldspeak. It returns the node that was found and takes in a hash table and the oldspeak. First, the variable lookups is incremented because the hash table is being traversed. The index of the hash table to search in is found by hashing the oldspeak. First, I calculate the index by hashing the oldspeak (with the salt) (and modding it by the hash table's size). Then, I return the node that results from a call to bst_find(), with the BST at the calculated index in the hash table's trees array and oldspeak as parameters.

The function ht_insert() is used to insert a node into the hash table that contains the specified oldspeak. Its return type is void, and takes in a hash table and the oldspeak. First, the variable lookups is incremented because the hash table is being traversed. The index of the BST (in the hash table) to insert in is found by hashing the oldspeak. First, I calculate the index by hashing the oldspeak (with the salt) (and modding it by the hash table's size). Then, I call bst_insert() on the BST at the calculated index in the hash table's trees array (passing in the BST and oldspeak). I set the result of the call to bst_insert to the BST at the specified index because it will "update" the root of the tree if necessary when a node is inserted.

```
double ht_avg_bst_size(HashTable *ht):
    int sum = 0;
    for i in range(0, ht_size(ht)):
        sum += bst_size(ht.trees[i])
    #end for
    return (double) sum / ht_count(ht)

double ht_avg_bst_height(HashTable *ht):
    int sum = 0;
    for i in range(0, ht_size(ht)):
        sum += bst_height(ht.trees[i])
    #end for
    return (double) sum / ht_count(ht)

void ht_print(HashTable *ht):
    for i in range(0, ht.size):
        node_print(ht.trees[i]
    #end for
```

The functions ht_avg_bst_size() and ht_avg_bst_height() are used for tracking statistics about the BSTs in the hash table. They each return a double and take in a HashTable pointer. Both of their structures are similar. I iterate through the trees of the HT, adding to a sum variable with each of the BST's size or height respectively. Then, I return the result of division between the sum and the ht_count() of the hash table. This result is cast as a double because that is what the function returns.

The function ht_print() prints the hash table. I iterate through the hash table's trees array, and I print each BST in it using node_print() on the contents of trees at index i.

speck.c/parser.c:

These files will not be altered at all, but they are a significant part of this program. The speck.c file contains the implementation of the SPECK hash function, that will be used for both the bloom filter and the hash table. The file parser.c contains the two functions next_words() and clear_words(). The function next_word() is used to parse through the input; it finds the next words in the input that matches the specified regular expression. The function clear_words() clears out the static word buffer. Both of these functions are used in banhammer.c.

Regex:

The regex expression that we are using in this assignment is for determining what a word is. If a word from stdin is matched by the regex expression that we specify, then we can check if it is in the bloom filter and hash table. A valid word (per our regex expression) must be a sequence that contains one or more characters of a set. It must have one or more letters, lowercase or uppercase, or numbers, or underscores. The expression must also handle words with contractions and hypenations.

Regex expression used: [A-Za-z0-9_]+(('|-)[A-Za-z0-9_]+)*

The initial part of the expression, [A-Za-z0-9_]+, means that the word must contain one or more letters (uppercase or lowercase), numbers, or underscores. The next part is a grouping enclosed with ( ) and ending with a *, because what occurs after the first section can occur zero or more times (not necessarily present). Within the grouping, I have ('|-), signifying that a dash or apostrophe could occur between two parts of a word. The next part, [A-Za-z0-9_]+, is the same as the first, meaning that a letter, number or underscore is present one or more times after the dash or apostrophe. This regex expression (defined as WORD in banhammer.c) will encapsulate all possible words that can be checked for violations.

banhammer.c:

This file is the main file and will be used to parse command line options and print the necessary outputs. First, I include all the header files, define the command line options, and define the regular expression.

```
#include all headers

define OPTIONS
define WORD

int main(int argc, char **argv):
   set booleans and default sizes
   int opt = 0;
   while ((opt = getopt(argc, argv, OPTIONS)) != -1):
     switch (opt):
        for all cases:
           set boolean to true or set size = strtoul(optarg, NULL, 10) (if valid)
           break;
     #end switch
   #end while

   if help specified:
      print help message

   if (bf_size >= 0):
      print error and exit
     if (ht_size >= 0):
      print error and exit

   create BloomFilter bf with specified size
   create HashTable ht with specified size

   badsp = opened badspeak.txt
   newsp = opened newspeak.txt
   create buf for bad words, old words, and new words

   while (scanning words from badsp and store into bad_buf):
      bf_insert(bf, bad_buf)
      ht_insert(ht, bad_buf)
   #end while
   close(badsp)

   while (scanning words from newsp and store into old_buf and new_buf):
      bf_insert(bf, old_buf)
      ht_insert(ht, old_buf, new_buf)
   #end while
   close(newsp)
```

Within the main function, I set booleans for command line options and the default values for the size of the bloom filter and hash table (2^16 and 2^20). I parse the command line options using a getopt loop. Within the switch statement, for statistics and help (-s and -h), I set the corresponding booleans to true. For the bloom filter and hash table size (-f and -t), I set them to what was inputted after the CLO (if it was valid); otherwise, I keep the default values. Then, if the help message option was specified, I print it. I also check if the bloom filter and hash table size are valid (>0) and exit if they aren't.

I create a bloom filter and hash table with the specified size. The files badspeak.txt and newspeak.txt contain words that are violations, so I have to add those words to the bloom filter and hash table. I open badspeak.txt and newspeak.txt and create buffers to read in bad words, words that have translations (oldspeak) , and those translations (newspeak). I use fscanf until the end of each file is reached, reading each word into the necessary buffers, and then insert the words into the bloom filter and the hash table. Then, I close the badspeak and newspeak files.

```
create regex_t re
if re did not compile:
   print error and return 1

char *word = NULL
create BST for bad words (bad_bst)
create BST for mixed words (mixed_bst)
while (getting VALID words from stdin with next_word()):
   iterate through word and make each char lowercase with tolower()
   #end for
   if word is in bloom filter:
      Node *check = lookup of the word in the hash table
      if (check && check.newspeak != NULL):
         insert check into mixed_bst with bst_insert()
      else if (check && check.newspeak == NULL):
         insert check into bad_bst with bst_insert()
      #end if
   #end if
#end while
clear_words()
free regex_t

if (stats):
   print stats
else:
   if bad_bst and mixed_bst contain words:
      print mixspeak msg
      bst_print(bad_bst)
      bst_print(mixed_bst)
   else if only bad_bst contains words:
      print badspeak msg
      bst_print(bad_bst)
   else if only mixed_bst contains words:
      print goodspeak msg
      bst_print(mixed_bst)
#end if-else

delete mixed_bst and bad_bst
delete hash table and bloom filter
return 0
```

Cite: some lines for parsing with regex are from section 8 of the assignment pdf

Then, the regex expression needs to be used. I create a regex_t to be used along with the defined expression in next_word(). I also create 2 different binary search trees (bad_bst and mixed_bst) to store words, one for bad words and one for words with translations.

Then, I read the words from stdin using the function next_word() in a while loop. Within the loop, the first thing I do is convert the current word to lowercase because all of the words in badspeak and newspeak.txt (in the bloom filter and hash table) are lowercase. If the word is in the bloom filter (bf_probe() on the word returns true), then I check the word further. I create a node pointer called check that is the result of a call to ht_lookup() on the current word from stdin. If check isn't NULL and its newspeak is not NULL (the word was found in the hash table and has a translation), I insert the word into the BST I created for words with translations with bst_insert(). If check isn't NULL and its newspeak is NULL (the word was found in the hash table and does not have a translation), I insert the word into the BST I created for "bad" words with bst_insert(). Then, I call the function clear_words(), to clear memory. Finally, I print the necessary output. I print the statistics if specified; otherwise, I print the corresponding message. If both the trees I created to store words have contents, I print the mixspeak message and print both messages. If only the tree storing "bad words" has contents, I print the badspeak message and that tree. Finally, if only the tree for translated words has contents, I print the goodspeak message and that tree. The last thing I do is delete the trees, delete the bloom filter and hash table, and return 0.