Mika Peer Shalem

Asgn 6

Final Design Doc

Assignment 6: The Great Firewall of Santa Cruz: Bloom Filters, Linked Lists and Hash Tables

General Overview

The assignment would introduce concepts on bit vectors, bloom filters, linked lists, and hash tables. We were chosen as the new leader and wanted to employ censorship on the citizens of the country. The assignment also includes new vocabulary: oldspeak means words that should not be used, newspeak is the correct translation of these words, and badespeak is oldspeak without its newspeak correlation. We would enter the badspeak to the bloom filter, while adding the the pairs to the hashtable.

banhammer.c

Main function:

```
Set the default value of the variable hash_size to 10,000
Set the default value of the variable bloom_size to 2^19
Set the default value of the variable mtf to false
While getopt argument is not -1 (getopt goes through all the given options and returns -1 when it runs out of options):

    If the argument is equal to h:
        Displays help message using the function
        Exit with 0
    If the argument is equal to t:
        Convert optarg to an int and set hash_size to optarg
    If the argument is equal to f:
        Convert optarg to an int and set bloom_size to optarg
    If the argument is equal to m:
        Set mtf to true
    If the argument is equal to s:
        Enables printing statistics to stdout
    Else
        Displays help message using the function
        Exit with a nonzero number


// initalize functions
Initialize the bloom filter and hash table using the sizes given above
Read a list of badspeak from a provided text file and insert it in the bloom filter
Read a list of old and newspeak pairs, insert the badspeak to the bloom filter, and insert the
oldpspeak to the hash table

// figure out which crimes the citizen did
While there are more unprocessed words in standard input:
    Use parsin module to read a word
    If a word was added to the bloom filter:
        If the word is found in the hashtable AND doesn't have a newspeak translation:
            Set crime to thoughtcrime
            Add the badspeak word to the crime_array
        If the word is found in the hashtable AND has a newspeak translation:
            Set crime to rightspeak
            If the badspeak is not on the array:
                Add the badspeak and newspeak words to the crime_array

// Print a warning message based on the crime:
If the user didn't enable stats:
If did both:
    Print the message based on a format shown on the assignment pdf
    Send citizen to joycamp
If only thoughtcrime:
    Print the message based on a format shown on the assignment pdf
    Send citizen to joycamp
```

If only requires conseling:

Print the message based on a format shown on the assignment pdf

If user enables stats:

Calculates bits examined per miss, false positives, bloom filter load, and average seek length based on the instructions on the assingmemnt pdf:

$$\text{Bits examined per miss} = \frac{\texttt{bf\_bits\_examined} - \texttt{N\_HASHES} \times \texttt{bf\_hits}}{\texttt{bf\_misses}}$$

$$\text{False positives} = \frac{\texttt{ht\_misses}}{\texttt{bf\_hits}}$$

$$\text{Bloom filter load} = \frac{\texttt{bf\_count()}}{\texttt{bf\_size()}}$$

$$\text{Average seek length} = \frac{\texttt{ht\_examined}}{\texttt{ht\_hits} + \texttt{ht\_misses}}$$

Print the calculated values with 6 digits of precision

bv.c

Goal: Implementation of bit vector ADT

BitVector *bv_create(uint32_t length)

Goal: The constructor for a bit vector. Returns a pointer to the bitvector if can allocate the memory, and NULL if not.

Allocate memory for the bit vector using malloc and the size of a BitVector structure.

Malloc either returns NULL if the memory cannot be allocated or a pointer to the allocated bit vector.

If the pointer is not NULL:

Allocate memory for a vector using the argument *length* as the size of the vector. We will dynamically allocate it using the *calloc* because we want each bit to be initialized to 0.

Set the length member to the length argument

Return the bit vector

void bv_delete(BitVector **bv)

Goal: The destructor for a bit vector.

```
If the pointer to bv exists:
    Free the vector member that is associated with the pointer to the argument bv
    Free the bit-vector bv
    Set the pointer to bv to NULL
```

uint32_t bv_length(BitVector *bv)

Goal: Returns the length of a bit vector.

```
Access the length member that is associated with the argument bv and return it
```

void bv_set_bit(BitVector *bv, uint32_t i)

Goal: Sets the i th bit in a bit vector.

```
If bv is not NULL //exits
    Locate the byte by doing i / 64
    Locate the bit by doing i % 64
    Access the location of the vector by doing masking and shifting
```

Ex.

Value: 11**0**0 1010, want to set the third bit

Byte = 3/64 = 0

Bit = 3 % 64 = 3

Location = 64*0 + 3 = 3

void bv_clr_bit(BitVector *bv, uint32_t i)

Goal: Clears the i th bit in the bit vector.

```
If bv is not NULL //exits
    Locate the byte by doing i / 64
    Locate the bit by doing i % 64
```

Access the location of the vector by doing masking and shifting

Ex.

want to clear the fourth bit

Byte = 4/64 = 0

Bit = 4%64 = 4

Location = 64*0 + 4 = 4

uint8_t bv_get_bit(BitVector *bv, uint32_t i)

Goal: Returns the i th bit in the bit vector.

Locate the byte by doing i / 64

Locate the bit by doing i % 64

Move a 1 bit to the right positive using the bit variable, then & it with the byte, and shift back

Return the the value of the vector

void bv_print(BitVector *bv)

Goal: A debug function to print a bit vector.

```
A loop that goes from var=0 to var<length of the bit vector, where the var increases by 1:
    Print the value at the vector member that is associated with the bit-vector at the location
    of var
```

bf.c

Goal: Implementation of bloom filter ADT. Bloom filter takes an object and says if it is definitely not in the array or if it's possibly in the array.

void bf_delete(BloomFilter **bf)

Goal: The destructor for a Bloom filter.

```
If the pointer to bf exists:
    Call the bit-vector function bv_delete that was implemented in the bit-vector structure on
the member filter.
```

```
        Free the bloom-filter bf
        Set the pointer to bf to NULL
```

uint32_t bf_size(BloomFilter *bf)

Goal: Returns the size of the Bloom filter.

```
Access the member filter of the bloom filer
Call the bv_length that was implemented in the bit-vector structure
Return the value
```

void bf_insert(BloomFilter *bf, char *oldspeak)

Goal: Takes oldspeak and inserts it into the Bloom filter.

```
// Oldspeak would be saved in 5 different bits in the array. All of these would be 1.
A loop that goes from i=0, to i<N_HASHES, where i increases by 1 in each iteration:
        Call city hash with the values salt[i] and oldspeak and save the value in a variable
        Call the bv_set_bit on the bit-vector filter with the variable that hold the bit location
Increase the value of n_keys by 1
```

bool bf_probe(BloomFilter *bf, char *oldspeak)

Goal: Probes the Bloom filter for oldspeak.

```
A loop that goes from i=0, to i<N_HASHES, where i increases by 1 in each iteration:
        Call city hash with the values salt[i] and oldspeak and save the value in a variable
        Call the bv_get_bit function on the bit-vector filter with the variable that hold the bit
location
        If the bit is equal to 0:
                Increase n_misses
                Return false
Increase the value of n_hits by 1
Return true
```

uint32_t bf_count(BloomFilter *bf)
Goal: Returns the number of set bits in the Bloom filter.

Go through all bits in the filter member, and count the number of bits that are set to 1

void bf_print(BloomFilter *bf)
A debug function to print out a Bloom filter.

Prints the values of all members: n_keys, n_hits, n_misses, n_bits_examined, the filter, and the 5 salts

bf_stats(BloomFilter *bf, uint32_t *nk, uint32_t *nh, uint32_t *nm, uint32_t *ne)
It sets each passed pointers to the value of the original variable for the relevant statistic that the Bloom filter is tracking

Set *nk to n_keys
Set *nh to n_hits
Set *nm to n_misses
Set *new to n_bits_examined

node.c

Goal: Implementation of node ADT

My own strlen function:

```
If the string exists
While we are not at the end of the string (in C, strings end with /0):
        Increase the value of a count variable
Return count
```

My own strdup function:
Goal: duplicate a string array

```
Create a new array with the size of the passed argument
While we are not at the end of the string (in C, strings end with /0):
    Get the next character in the string array
    Set the index of the new array to the character
Return the duplicate array
```

Node *node_create(char *oldspeak, char *newspeak)

Goal: The constructor for a node.

```
Allocate memory for a node using the size of the node structure and malloc since its values do not matter
If the allocation of memory was successful:
    Set the oldspeak member of the new node to the return value of my strdup function with the oldspeak argument
    Set the newspeak member of the new node to the return value of my strdup function with the newspeak argument
    Set the value of the next and prev members to NULL
```

void node_delete(Node **n)

Goal: The destructor for a node.

```
If the pointer to n exists:
    Free the node n
    Free the oldspeak and newspeak members
    Set the pointer to n to NULL
Free the memory used by oldspeak and newspeak
```

void node_print(Node *n)

Goal: prints the contents of the node. Used to produce the correct program output.

```
If the value of the members of oldspeak and newspeak are not NULL:
    Print both of them using the format given in the assignment pdf – printf("%s -> %s\n", n->oldspeak , n->newspeak);
If the value of newspeak is NULL:
    Print only oldspeak using the format given in the assignment pdf – printf("%s\n", n->oldspeak);
```

ll.c

Goal: Implementation of linked list ADT. Used to help resolve hash coli

Using the same strlen implementation that is described above.

My implementation for strcmp:

```
Goal: compare two strings, return true if they are the same and false if they are not.
If the lengths of both string are different:
    Return false
Loop through all characters in the strings:
    If they are different, return false
Return true
```

LinkedList ll_create(bool mtf)

Goal: The constructor for a linked list.

Dynamically allocate space to the linkedlist using malloc since it does not matter what the intialized values are. The size is the size of a linked list structure.

```
If the memory was successfully allocated:
    Set the length member to 0
    Set the prev member of the head node to the address of the tail node, and the next to NULL
    Set the next member of the tail node to the address of the head node, and the prev to NULL
    Set the member mtf to the value of the argument
```

void ll_delete(LinkedList **ll)

Goal: The destructor for a linked list.

```
While the pointer to the head member is not NULL:
    Set the pointer to the next node to NULL
    Set next to the value that the head pointer points at
    Delete each node by calling node_delete on the head
    Set the pointer of the head to next
Free the linked list and set it to NULL
```

uint32_t ll_length(LinkedList *ll)

Goal: Returns the length of the linked list, which is equivalent to the number of nodes in the linked list, not including the head and tail sentinel nodes.

```
Return the value of the length member of the linked list
```

Node *ll_lookup(LinkedList *ll, char *oldspeak)

Goal: Searches for a node containing oldspeak. If a node is found, the pointer to the node is returned. Else, a NULL pointer is returned. If a node was found and the move-to-front option was specified when constructing the linked list, then the found node is moved to the front of the linked list.

Make sure to count the number of seeks and links! Seeks increases by 1 whenever ll_lookup is called, and linked increases by 1 every new iteration of the loop

```
Go through all nodes in the list:
    If the oldspeak member of the node is not equal to the oldspeak argument:
        If the member mtf is true:
            // make sure all the nodes are still connected:
            Set the previous node's next to the address of the current node's next
            Set the next node's previous to the address of the current node's previous
            // move the node to the front
```

```
                Set prev to the head
                Set next to head->next
                Set head's next node to the new node
        Return the pointer to the node
    Else:
        Return NULL
```

void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak)

Goal: Inserts a new node containing the specified oldspeak and newspeak into the linked list.

```
Call the ll_lookup function that we implemented.
If the result is NULL: // oldspeak is not in the list
    Create a new node using:
        The argument oldspeak
        The argument newspeak
        Set next to the address that head's next points at
        Set prev to the address of the current head
    Set head's next node to the new node
    Increase the length member by 1
```

void ll_print(LinkedList *ll)

Goal: Prints out each node in the linked list except for the head and tail sentinel nodes.

```
Go through each node of array (from the second one and stopping before the tail):
    Call the function node_print that is described above
```

void ll_stats(uint32_t * n_seeks, uint32_t * n_links)

Goal: Copies the number of linked list lookups to n_seeks, and the number of links traversed during lookups to n_links.

```
Set n_seeks to seeks
Set n_links to links
```

My idea for traversing a linked list:

```
Set temp to head
While (temp -> next is not the tail) // while the next node exists
    … do actions
    Set temp to next
```

Test ideas for linked list delete:

- Check that after the link is deleted, all memory is freed using valgrind (checks ll_delte)
- Try to access one of the members after running ll_delete → supposed to do a seg fault

ht.c

Goal: Implementation of hash table ADT

void ht_delete(HashTable **ht)

Goal: ht_delete is the destructor for a hash table.

```
If the pointer to ht exists:

    Go through all the lists in the hash (number of lists is saved in the var size):
        Delete the linked list using ll_delete()
    Free the lists member of the hash table set its pointer to NULL
    Free the hash-table ht
    Set the pointer to ht to NULL
```

uint32_t ht_size(HashTable *ht)

Goal: Returns the hash table’s size.

Access the size member of ht and return it

Node *ht_lookup(HashTable *ht, char *oldspeak)

Searches for an entry, a node, in the hash table that contains oldspeak. A node stores oldspeak and its newspeak translation. The index of the linked list to perform a look-up on is calculated by hashing the oldspeak. If the node is found, the pointer to the node is returned. Else, a NULL pointer is returned.

```
Call ll_stats to properly track the number of links.
    Call once at the beginning of the function, and once after calling ll_lookup
    Find the difference between them and set it the n_examined member of the hash table

Go through all the linked lists in the hashtable ht:
    Call hash (the CityHash function) with the salt member of ht and oldspeak
    Call ll_lookup with the oldspeak // pointer to the location of oldspeak or NULL
    If the return value of the linked list lookup is NULL:
        Increment the value of n_misses by 1
        Return a NULL pointer
    Else:
        Increment the value of n_hits by 1
        Return a pointer to the node
```

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)

Goal: Inserts the specified oldspeak and its corresponding newspeak translation into the hash table.

```
Call hash (the CityHash function) with the salt member of ht and oldspeak
If the list at that index is already intialized:
    Use ll_insert with the oldspeak and newspeak argument
Else:
    Create a linked list by passing the mtf member
    Use ll_insert with the oldspeak and newspeak argument
Increase n_keys by 1
```

uint32_t ht_count(HashTable *ht)

Returns the number of non-NULL linked lists in the hash table.

```
Go through all the linked lists in the hash table ht:
    Count the number of non-NULL linked lists
Return the count
```

void ht_print(HashTable *ht)

Goal: A debug function to print out the contents of a hash table. Write this immediately after the constructor.

```
Go through all the linked lists in the hash table ht:
    Use the ll_print function to print the contents of the linked lists
Print the contents of all members of ht (salt, size n_keys, n_hits, n_misses, n_examined, and mtf)
```

void ht_stats(HashTable *ht, uint32_t *nk, uint32_t *nh, uint32_t *nm, uint32_t *ne)

Goal: sets *nk to the number of keys in the hash table, *nh to the number of hits, *nm to the number of misses, and *ne to the number of links examined during lookups.

```
Set nk to n_keys
Set nh to n_hits
Set nm to n_misses
Set ne to n_examined
```

parser.c

Goal: Implementation of the parsing module

Parser *parser_create(FILE *f)

Goal: The constructor for the parser.

```
Allocate memory needed for a single parser object using malloc and the size of a parser object.
If the memory was successfully allocated:
    Initialize a parser object with file f, current line equal to 0, and line offset equal to 0
Return the parser
```

void parser_delete(Parser **p)

Goal: The destructor for the parser.

```
If the pointer to p exists:
    Free the file member that is associated with the pointer to the argument p
    Free the parser p
    Set the pointer to p to NULL
```

bool next_word(Parser *p, char *word)

Goal: This function will take a pointer to a parser and a buffer for the returned word.

The buffer holds the input and output variables (in this case, words). It is a var that takes input and sends output.

```
// Debug test: what is the 1000 character is in the middle of a word?
If this is the beginning of a line:
    Gets the next sentence and save it in the current_line member of the p structure
    Set the last value to '\n'
    Reset the line_offset to 0
    Go through all characters in the line:
        Get a character based on the index each time
        If it's a letter, number, -, or ':
            If it’s a letter:
                Convert to lower case
            Set the index of the word buffer tot he character
        Else:
            Break;
    Check that the word is valid (not empty, a space, or \n):
        Move the file pointer to the line offset
        Return true
If it’s not a new line:
    Do the same thing as described above, BUT don’t get a new line and don’t reset the line pointer

Return false
```

Makefile

Goal: compiles the program
Set the compiler to clang
Set the C flags to the regular flags used to compile the program -Wall -Werror -Wextra -Wpedantic

Define the target ‘all’ with the dependency of the file holding the main function (banhammer)

Have a variable that will hold all c files
Have a variable that will hold all object files (Use the pattern match placeholder % to covert the C file to an object file).

Define the target clean with the command to remove all compiler-generated files except the executables

Define the target spotless with the command to remove all files that are generated and the executable files. This will call ‘make clean’ to remove everything except executables, and then will remove the executable

Define the target format with the command to format the source files