Vincent Siu
11/20/22
Vsiu2
asgn5

Final Design

Goals: to implement 4 different data types that will be used to implement the Great Firewall. The Great Firewall is used to filter out any badspeak words or oldspeak words that each of the citizens uses. This design will include an outline for implementations of 4 different ADT's: bloom filter, bitvector, linked lists, and hash tables.

Implement my own version of str copy here.

Implementation for the bit vector data structure.

- Define the structure for the bit vector including the length and the vector pointer.
    - Length will define the number of bits
    - Pointers will be defined dynamically representing each of the uint64t's.

- Implement the function to create the bit vector. Length defines the number of bits. This will be the constructor.
    - Using malloc to allocate appropriate amounts of memory for bit vectors, use sizeof(bitvector).
    - If bit vector is successfully created
        - Make a copy variable that will store the value of the length variable.
        - While the copy variable is more than 64 in length
            - Add 1 more uint64 to the vector. Use a variable to store the amount to be added.
            - Then for cases where the copy variable is less than or equal to 64, add one more vector. This is to make sure that we have enough bits to represent the desired length.
            - Use calloc to create the bit vector. We use calloc because we must set all bits to 0 when creating the vector.
            - If calloc fails, or the vector fails to create, return null
        - Return the bitvector pointer.
- Implement a function to print the bit vector.
    - For loop through the bit vector until we reach bv->length to print the bit vector
- Implement the function to delete/destroy the bit vector
    - If the bv is defined or not NULL
        - Free the vector property then free the pointer.
- Implement the function to get the length of the bit vector.
    - Return the length, already given in the struct!
- Implement the function to set a bit within the bit vector. This should make ith bit as specified equal to 1.
    - Get the position of the bit using %64 and then use /64 to get which array in which the bit where located must be set.

    - Create another set of bits starting with 1 at the right most and zeros everywhere else and shift it to the desired position, should use left shift.
    - Or these bit vectors together.
- Implement the function to clear the bit. This should make the ith bit as specified equal to 0
    - Get the position of the bit using %64 and then use /64 to get the array in which the bit located must be cleared.
    - Create another set of bits with a 1, then bit shift that to the desired position.
    - Do the AND INVERSION of these two bits in order to properly clear it!
- Implement the function to get the ith bit in a vector. This should return a uint8_t.
    - Get the position of the bit using %64 and then use /64 to get the array in which the bit located must be tested.
    - Create another set of bits with a 1, then bit shift that to the desired position.
    - And these two two bits together
    - Get the result and shift THAT TESTED bit by right shifting.
    - Return THAT result.

Implementation of the bloom filter data structure
- Define all the default salts that will be used for hashing to create an initialization key.
- Define the structure for bloom filter, which will consists of:
    - Salts. This will be used to create an initialization key. In this case, k=5 different salts will be used.
    - N_keys. To task the number of keys that input into the bloom filter.
    - N_misses. To track the number of probes that return false.
    - N_bits_examined. This will track the total number of bits examined. In this case, it will range from 1-5 fo each probe.
    - A bit vector data type. This bloom filter will utilize a bit vector.
- Implementation to create the bloom filter.
    - Allocate memory for the bloom filter.
    - If successfully created
        - Initialize bloomfilter’s properties, being keys, n_bits, n_misses, n_bits_examined to 0.
        - Initialize the salts array to the default salts as defined above. This will initialize the initialization keys.
        - Create a bit vector that will be used as a filter for the bloom filter itself.
        - If the bit vector was not successfully created
            - Free the pointer bf is in
            - Set bf = NULL
        - Return bloom filter
- Implementation to destruct the bloom filter.
    - While the bf is not equal to null
        - If the filter property (the bit vector) is defined, then delete the bit vector using its delete function.

        - Free the pointer and set to null as necessary.
- Implementation to get the size of the bloom filter. This should return the number of bits that the bloom filter can access.
    - Return the length as given above.
- Implementation to insert an oldspeak into a bloom filter. This should accept the bloom filter to modify as well as the old speak to add (type char).
    - Looping through each salt, hash the word to each of them using the hash function from city hash, setting the bits in the bit vector to represent the binary values of the oldspeak.
    - Increment the number of bloom filter keys.
- Implementation to probe bloom filters for old speak, returning T or F based on whether or not an element is in the bloom filter.
    - Similar to insert, loop through the salts array, and hash each with the oldspeak
    - Check if the bits have been set based on the has value given using get_bit(), return true (which means that oldspeak was most likely added to the bloom filter), or otherwise return false.
- Implement a return to the number of set bits in the bloom filter.
    - Set a variable to track down the number of bits set = 0.
    - accessing bf - > bitvector structure
        - Loop through each of the bits using get_bit(iterator)
            - If get bit equals 1, then add 1 to the bit counter
    - Return the counter.

Implementation for Node. This will define the points that point from one thing to the other, like the old speech pointing to the newspeak and vice versa.

- We will need to use a function to copy a string.
    - For each index while the current index of the string or char array is not null and index value is less than the strlen, set the destination index of where it is being copied to.
    - Null terminate string, so we don't have more unwanted characters after all valid characters
- We will need to make a str len function
    - Define a variable to track the length
    - While the current length of the index of the string is not null
        - Add to the length variable.

- Note: we do not need to define the structure of the node since it's defined in the header files. We will need to use its properties outside of node.c

- Define the constructor for the node
    - Allocate memory for the node struct. If all goes successfully, do the following.
        - Get the length of the string using strlen written above.

        - Memory allocate, using calloc the word that the oldspeak being passed in is going to be copied to, which should be node's oldspeak property. Then copy to this newly created word using my strcopy.
        - If the newspeak != NULL, do the exact same thing with oldspeak, but using the newspeak in-place for the oldspeak. If it is NULL, set it to NULL.

- Define the implementation to delete a node, for example, we will call the node to be deleted, n
    - If n is not null
        - Free the oldspeak and newspeak if neither of these are null.
        - Free (n) pointer.
        - Set n = NULL

- Define the implementation to print the node
    - If the newspeak is null
        - Only print oldspeak
    - Else
        - Print oldspeak and newspeak.

Next, we must implement the linked list data structure that will be used along with the hash table.

- This will include the length of the linked list (uint32_t), usage of the Nodes we implemented earlier (head and tail), and mtf.
- We will need to write an str compare function
    - If both strings are not null, use a while loop to get the length of both strings.
    - If the lengths are not equal, its obvious to us that the strings do not match and return false in this case
    - Loop through both of the strings and check each of their indices and if any one of them do not match, automatically return false.
- Implement the constructor of the linked list. The parameter it takes is move to front, meaning if the node at the tail end goes back to the head or not.
    - Allocate memory for the struct of the linked list. If all goes successfully then do the following.
        - Using node create, we have to create the head and tail nodes.
        - Set head's next value to tail
        - Set tail's next value to null
        - Set tail's previous value to head
        - Head head's previous to null.
        - Initialize the seeks and links to 0.
        - Return the linked list pointer.
    - Return null if it failed to memory allocate.
- Implement the destructor of the linked list.
    - Set two nodes

        - One will represent the current node and the other will represent a copy of it for the curr node value to be preserved.
        - If linked list is defined
            - The curr node will be the head of the linked list and set t = to curr
        - While curr is not equal null
            - Set t = curr to preserve the value of curr
            - Delete the curr node
            - Set curr = to curr (or t) next.
        - Null out the original linked list pointer.
- Implement a function to find the length of the linked list.
    - Add a variable to track the counter
    - Walk through the linked list. While the node next value is not equal null
        - Add to the counter.
    - Return the counter - 2, so we don't count the head and tail nodes. The entire linked list was walked though.
- Implement a function to lookup a node in a linked list.
    - Walk through the linked list, while the head node is not equal null (while we have not reached the end of the linked list).
        - Add the number of links.
        - Set the current temp node equal to what is next in the linked list value.
        - Use strcmp function, as written above, to compare our passed in value to the current node's oldspeak.
    - If the mtf set to true
        - Set the previous node to the next
        - Do vice versa as well
        - Set the current node's next equal to the first node to the linked list
        - Set previous equal head
    - Return the current node where it was found. If not found, return null.
- Implement a function to insert a node into the linked list.
    - First we must check if the value to insert already exists. Call lookup with the value to insert and if found, return
    - Otherwise insert it into the beginning of the linked list where its next value = to the previous first value and its value equals head.
- Implement a function to print the linked list
    - While we are not at the tail end of the linked list, beginning at the first valuelist except for the head and the tail
        - Call print_node, throughout the while loop, this will print the whole linked list

Vincent Siu
11/20/22
Vsiu2
asgn5

Parsing from a file

- Write a function to compare chars. It will take in a char pointer, a char, and an int length.
    - Assuming that we don't go out of bounds, if the current indice of what's being compared is equal to what we have passed in return true or return false otherwise.
- Define the structure for the parser
    - Define the FILE object type
    - Define a char that represents the current line, an array of chars, which must be at most 1000 characters.
    - Define a line offset that will be used to tell where we left off if there are more valid words.
- Parser create constructor which will take in the file pointer stream
    - Allocate memory for the parser. If not allocated, then set it to null and return null.
    - Set parser's file object to whatever file pointer was passed in
    - Initialize offset.
    - Return the parser pointer.
- Implement a destructor for the parser
    - Free parser and set to null if the pointer exists.
- Next word function. This function will take a parser and a word line to parse out any valid words within the line
    - If the offset is equal 0, then copy over the word that came from fgets into the current line array. During this process, we set each character to lower case using tolower().
    - Initialize a boolean variable, valid character found, to be false.
    - Set a variable iterator
    - While the current line array, indexed at the variable iterator is not null
        - Check if current character is alphanumeric or is a dash or single quote using the str compare function
            - In the event a valid character is found, set valid character found equal to true.
            - Copy over the current character to word (what was passed in).
            - Increment index iterator.
        - Else if we have come to an invalid character and we have already came across valid characters
            - Stop looping
        - Increment offset by 1
    - Outside of while loop, since we have stopped, null terminate the word.
    - If we have not found a valid character and the current character of current line indexed at offset is equal to null
        - Reset null and reset the current line array
        - Return false.
    - Return the bool valid character found

Vincent Siu
11/20/22
Vsiu2
asgn5

Implement the hash table

- Define a structure for the hash table
    - Similar to the bloom filter, we will need the following variables: salt, size, n_keys, n_hits, n_misses, n_examined. We need these to use as a stat to look up some oldspeak in a link list.
    - We will also need a bool for mtf or move to front.
    - Also need to define a linked list since this is a hash table.
- Define a constructor of the hash function specifying the size and whether or not the linked lists should be using the move to front or mtf.
    - Using malloc, appropriately allocate memory for the sizeof(hashtable).
    - Set ht mtf properties to whatever mtf passed is specified
    - Set the salt property to 0x9846e4f157fe8840
    - Set the n_hits, n_misses, n_examined, n_keys = 0.
    - Set the ht property size to whatever size specified was passed in
    - Set the ht property linked list to be calloced of a new linked list. Remember that we will be using link lists to link to a key in the hash table, so this is necessary,
    - If the linked lists was not successfully created
        - Free the hashtable pointer originally created
        - Set it equal to NULL
    - Return the hash table.
- Get the size of the hash table.
    - Return size property of the hash table.
- Implement a function to lookup a node in a hashtable. Search is performed based on the oldspeak.
    - Set two variables that will be passed by reference to track down being n_seeks and n_links. N_seeks tracks the number of linked list lookups and n_links tracks the number of links in a linked list we have looked through.
    - Call ll_stats with the two variables we have just initialized.
    - Use hash to get the hash value or index of the oldspeak
    - If the linked list array at the current index is defined
        - Call linked list lookup to seach for the node
        - Update llstats
        - If we have not found the node
            - Increment the number of misses
        - Else
            - Increment the number of hits otherwise.
    - Return the node.
- Implement insert method for the hash table data structure.
    - Call the hash function with the oldspeak
    - If the linked list at the current index does not have any instance of the oldspeak passed in, we may increment the number of keys.
    - If the current place has a NULL linked list, call ll_create

    - Insert the values into the link list. If there is no newspeak, set that value to null
- Implement ht_count: count the number of non null linked lists within the hash table
    - Set a variable to track the counter.
    - Loop through the hash table. If its linked list value is non-null, increment the counter.
    - Return the counter.
- Implement the hash table stats. Parameters passed into the function include number of keys, number of hits, number of misses, and number of links.
    - Update each of the hash table properties, being number of keys, number of hits, number of misses, and number of links examined during lookup
    - I.e. n_keys = nk.

Implement the main function, banhammer.

- Initialize the bool to print stats and mtc to false
- Initialize the size for bloom filter and hash table to be 2^19 and 10000 respectively.
- The option parser will accept the following options, h,t,f,m,s
    - h: to print help menu
    - t: set the size of the hash table
    - f: set the size of the bloom filter
    - m: move to front = true. This will allow the mtf operation in the linked list whenever a lookup is called.
    - s: print stats and suppress all other printed messages.
- For both badspeaks and newspeaks
    - While not at end of file,
        - Get the next line
        - While next word(line) returns true
            - Add oldspeaks/ badspeaks and newspeaks to the hash table and bloomfilter.
- Create two linked lists to hold the badspeaks and the oldspeaks
- Take in user input. While not at end of user input
    - Check of bloom filter if the citizen said something wrong and it it returns true
        - Check the hashtable using ht_lookup
            - If the node returned is not equal null, this means we have to take action
                - If the newspeak does not exist, add it to the badspeaks linked list. Add it to the old speaks otherwise.
- If oldspeaks linked list and badspeaks linked list are both bigger than 0 and print stats is disabled
    - Print the mixed speak message and print both linked lists.
- If the oldpeaks linked list is the only one bigger than 0 and print stat is disabled
    - Print goodspeak message and oldspeak linked list.
- If the badspeaks linked list is the only one bigger than 0 and print stat is disabled

  - Print badspeak message and badspeak linked list.
- If print stats is enabled
  - Initialize variables to print stats, should include ht keys, ht hits, ht probes, bf keys, bf hits, bf misses, bf examined, examined per miss, avg seek length, and bf load.
  - Call both ht and bf stats functions to get all of these values, since they are passed by reference.
  - Print all stats.