Assignment 6 Design - Jack Vento

This program, while long, has a rather simple core design. First, it passes "nonsense" words supplied from an external file through a bloom filter. Then, it checks a supplied list of words against this bloom filter. If the bloom filter finds them to be "nonsense" (i.e. searches and believes it finds them), then it will pass them along to a hash table. This hash table will then try to translate the nonsense word to newspeak. If it is successful, it will notify the user of their error and carry on. If there is no translation found, then the user is "sent to the dungeon" for supplying a forbidden word.

Pre Lab

Part 1

1) Write down the pseudocode for inserting and deleting elements from a Bloom filter.

To insert, we will simply run our key (a string) through our hash function three times, supplying a new predetermined salt each time to get a new value (this essentially functions as three different hash functions). Then, we take the output from our hash (an index) and set the specified bit in our bit vector filter.

- First = hash(primary, key) % length
- Second = hash(secondary, key) % length
- Third = hash(tertiary, key) % length
- if First is in range [i, length]
    - bv_set_bit(First)
- if Second is in range [i, length]
    - bv_set_bit(Second)
- if Third is in range [i, length]
    - bv_set_bit(Third)

To delete, we will do the inverse of above. After getting our indexes from our hash function and the respective salt, we will clear the corresponding bit in our bit vector filter.

- First = hash(primary, key) % length
- Second = hash(secondary, key) % length
- Third = hash(tertiary, key) % length
- if First is in range [i, length]
    - bv_clr_bit(First)
- if Second is in range [i, length]
    - bv_clr_bit(Second)
- if Third is in range [i, length]
    - bv_clr_bit(Third)

2) Assuming you are creating a bloom filter with m bits and k hash functions, discuss its time and space complexity.

Since hash functions execute in constant O(1) time, having k hash functions will cause insertion and probing to execute in O(k * 1) = `O(k)` time. The more hash functions we have, the amount of false positives should be

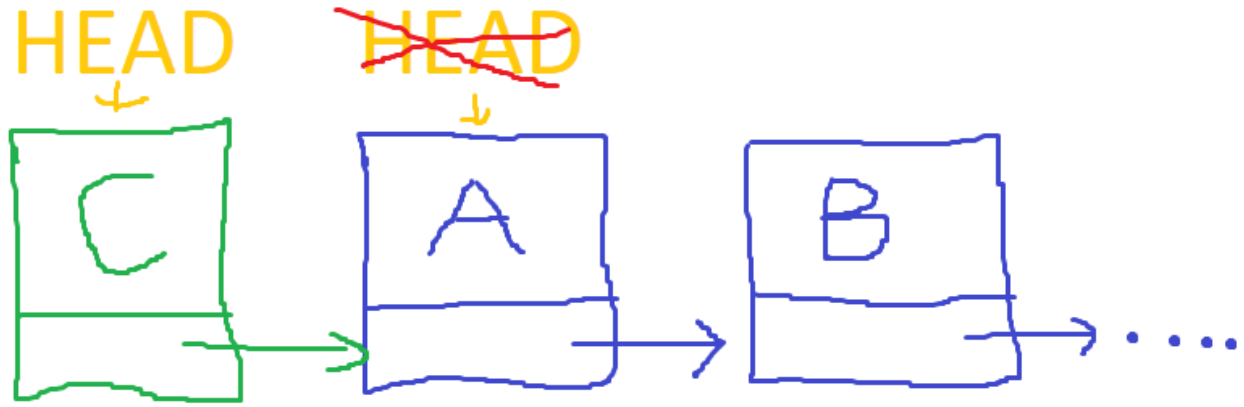
reduced, but the time complexity will unfortunately increase alongside that improvement. There are always trade offs.

As for space, since the size of our salts are always 3 * 128 bits, they are constant and not included in Big-O notation. Our bloom filter will scale linearly with size m bits (aka m/8 + 1 bytes in the bit vector), leading to a complexity of `O(m)`. Similarly to time, more bits will lead to less false positives (less collisions).

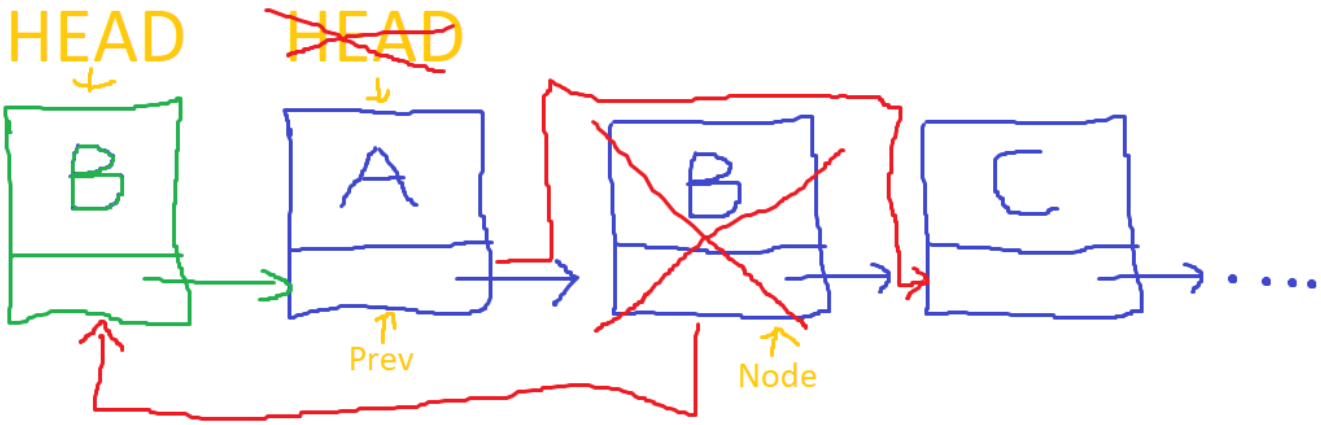
Part 2

1) Draw pictures to show how elements are being inserted in different ways in the Linked List.

Given a Linked List with element A at the head, we insert element C by creating a new node, setting its 'next' pointer to the current head, and then setting head to this new node like so:



For lookup insert (where a node is moved to the front when it's searched for), we must use two pointers: one to find the node (node) and one to keep track of the previous (prev) since this list is not doubly-linked. We will iterate both pointers through the list until we find the node we're looking for, keeping previous one node behind. Once we find the wanted node, we update prev's 'next' pointer to node's 'next' pointer to keep the correct order, set node's 'next' to the current head, and then finally set head to node.



2) Write down the pseudocode for the above functions in the Linked List data type.

Simple Insert:

- Create new node 'node' and set its data element

- Set node's next pointer to head
- Set head to node

Lookup Insert:

- if list is empty:
    - return null
- if head.data = node_to_find.data:
    - return head
- search = head.next, prev = head
- while search is not null:
    - if search.data = node_to_find.data:
        - if move_to_front = true:
            - prev.next = search.next
            - search.next = head
            - head = search
        - return search
    - prev = search
    - search = search.next
- return null

Bloom Filter

Design:

This bloom filter will help keep track of nonsense words by running a list of them through a hash with three different salts (resulting in three different hashes) and then setting the corresponding bits in a filter (a bit vector) to 1. This is an incrediblty efficient way to both search for and track nonsense words since hashing leads to constant operations and storing individual bits allows for incredibly efficient storing.

However, bloom filters (especially of smaller length) have significant problems with false positives due to hash collisions. So, searches cannot definitively say whether an item is present in a data set; it can only say whether an element is definitively *not* in a set. That is why they are often combined with hash tables to confirm positivity, which it is in this assignments. Probe (aka search) will use the bit vector helper functions to check if the corresponding hashes are set to 1.

For a discussion of complexity, see Prelab 1.2.

Pseudocode: Create(size):

- Allocate a bloom filter on the heap
- if allocation succeeded:
    - Store predetermined salts in the three salt arrays (primary, secondary, tertiary)
    - Allocate a bit vector on the heap of length *size*

Delete

- Free memory associated with the bit vector and bloom filter itself.

Insert(string key)

- For insertion, see Prelab 1.1.

Probe(string key)

- Calculate hashes using *key* for each of the salts and store them in respective integers: first, second, third
- if hashes are in range [0, size)
    - if bv_get_bit(first) = bv_get_bit(second) = bv_get_bit(third) = 1:
        - Return true
    - else:
        - Return false

Hash

Design:

This hash table will implement the same SPECK cipher to generate its hashes as the bloom filter. It will largely be a wrapper for accessing linked lists containing hatterspeak translations in constant time. Most of its functions rely on Linked List helpers for their output and the ADT itself does not do anything very notable besides finding and operating on the correct linked list.

Pseudocode: Create(length)

- Allocate hash table on the heap
- if successful:
    - Fill the salts with predetermined values
    - Set the length member variable to *length*
    - Allocate enough linked lists on the stack according to *length*

Delete

- Delete all of the linked lists, delete the array of heads, and then destruct the hash table itself

Count

- Return the number of entries in the hash table by iterating through the array of linked list heads

Lookup(string key)

- value = hash(key) % length
- head = heads[value]
- return ll_lookup(head, key)

Insert(HatterSpeak)

The insert will first look to see if the HT already contains the supplied HatterSpeak. If it does, it will attempt to add a translation if there is no existing translation using the hatterspeak string, then it will delete the supplied HatterSpeak (to avoid duplicate data). If there is no existing node at all, it will try to insert one.

- value = hash(HatterSpeak.oldspeak) % length
- head = heads[value] (Get head to insert at)
- node = ll_lookup(head, HatterSpeak.oldspeak)
- if node is not null (duplicate found):

        - if node.gs.hatterspeak is empty:
            - copy HatterSpeak into node.gs
        - Delete HatterSpeak
- else:
    - ll_insert(head, HatterSpeak)

Linked List

Design:

Rather than open addressing, this program uses linked list chaining for hash collisions. At each hash value, there is a singly-linked list containing HatterSpeak structs that have all the data necessary to translate words from oldspeak to hatterspeak. Depending on the status of the move_to_front boolean, the list will either move recently looked-up nodes to the front of the list or not. When this boolean is set to true, the linked list will take advantage of temporal locality: leading to significant performance improvements that help our hash table from decaying to linear time complexity. For insertion and lookup design/pseudocode, see Prelab 2.1-2.2.

Additionally, this file will also be the home of the HatterSpeak structure. It's a simple data type with only two members: oldspeak (a string of the uncensored word) and hatterspeak (a string of the translation).

Pseudocode: Create Node(HatterSpeak)

- Allocate node on the heap
- if allocation succeeded:
    - Set node data to HatterSpeak
- return node

Delete Node(Node)

- Delete Node.hatterspeak using hs_delete
- Delete the node itself

Delete List

- Create node pointers del and next
- while del is not null:
    - next = del.next
    - Delete_Node(del)
    - del = next
- head = null

Insert Node(HatterSpeak)

- See Prelab 2.2

Lookup Node(string key)

- See Prelab 2.2

HS (hatterspeak struct)

Design: HatterSpeak struct has its own file to help simplify memory management, avoid redundant code, and keep inclusions modular (and small). It is rather simple and consists entirely of the struct definition and two helpers (create/delete).

Pseudocode: Create Struct(string old, string hatter)

- Allocate size of struct on the heap
- oldspeak = allocate string size of old
- hatterspeak = allocate string size of hatter
- Copy old into oldspeak
- Copy hatter into hatterspeak

Delete Struct

- Delete oldspeak, hatterspeak strings
- Delete struct itself

Hatterspeak

Design:

After defining default values and global variables for statistics tracking, the program begins with a simple getopt loop that'll optionally update two booleans or overwrite one of three default values depending on what arguments are supplied. Optarg will contain the necessary information for overwriting defaults for -h and -f. I decided to go this route instead of an enum or array because I really only need to track two values: whether to suppress the censor letter and whether move-to-front has been previously specified (enforcing mutual exclusion between -m and -b).

The program will operate as follows:

1. Defines, global vars, and getopt loop.
2. Outside the getopt loop, a bloom filter, hash table, and two linked lists for output tracking will be created.
3. Open oldspeak.txt, create an input buffer, parse for input (forbidden words), and put them into lowercase.
4. For each forbidden word, a HatterSpeak struct without a translation will be created and added to the hash table with the oldspeak key inserted into the bloom filter.
5. Open hatterspeak.txt, create two input buffers, and parse for input (forbidden words with translations).
6. Parse the oldspeak-hatterspeak pairs, put them into lowercase, create HatterSpeak structs, and insert them into the BF + HT.
7. Compile a regex and parse text from standard input using that regex.
8. Create a new string buffer, use the parser/regex to find words, and put the words into lowercase.
9. Probe the Bloom Filter and if it was successful, confirm it wasn't a false positive by looking for it in the Hash Table.
10. If it is found, add it to either the nonsense (no translation) or hatterspeak (has translation) linked list.
11. Depending on whether -s was supplied, either calculate and print statistics using the global variables, or print letters + offending words depending on input.

Pseudocode:

Main

- Init booleans mtf_supplied = suppress_letter = false
- Initialize hash_size, bloom_size, and move-to-front to defaults.
- while there are options to parse:
    - switch option
        - s:
            - Set suppress_letter = true
        - h, f:
            - Update hash_size and bloom_size from defaults
        - m, b:
            - if mtf_supplied = false:
                - Set move_to_front to respective value
                - mtf_supplied = true
            - else: error and return
- Create bloom filter bf of size bloom_size
- Create hash table ht of size hash_size
- Created linked lists nonsense_words + hatterspeak_words
- Open oldspeak.txt in read mode
- Declare input buffer
- while words to parse in oldspeak.txt:
    - Convert word to lowercase
    - bf_insert(bf, word)
    - Create hatterspeak struct gs
    - gs.oldspeak = word
    - gs.hatterspeak = empty_string
    - ht_insert(ht, gs)
- Close oldspeak.txt
- Open hatterspeak.txt in read mode
- Declare two input buffers
- while pairs to parse in hatterspeak.txt:
    - Convert pair to lowercase
    - bf_insert(bf, word)
    - Create hatterspeak struct gs
    - gs.oldspeak = pair.first
    - gs.hatterspeak = pair.second
    - ht_insert(ht, gs)
- Close hatterspeak.txt
- Compile regex for parsing words
- while words to parse in stdin:
    - Convert word to lowercase
    - if bf_probe(bf, word) = true:
        - node = ht_lookup(ht, word)
        - if node is not null:
            - if node.gs.hatterspeak is null:
                - Add gs to nonsense_words

            - else:
                - Add gs to hatterspeak_words
- clear_words() to empty buffer
- Free regex
- if supress_letter = false:
    - Print letter
    - Iterate through nonsense_words and print
    - Iterate through hatterspeak_words and print
- else:
    - seek_avg = links_searched / seeks
    - list_avg = Amt_of_lists / ht_length
    - ht_load = ht_count / ht_length * 100
    - bf_load = bf_count / bv_length * 100
    - Print statistics
- Free nonsense_words & hatterspeak_words
- Free BF using bf_delete
- Free HT using ht_delete

Design Changes

Linked List

I originally overcomplicated the deletion, thinking I would need to keep the LL order in check upon deletion. After rereading, I found that not to be the case and simplified the function significantly.

HatterSpeak (struct)

While initially putting the struct in the Linked List header, I instead opted for its own file to avoid having the Linked List overhead where ever I needed to use the struct (nearly every file).

Hatterspeak (file)

I did not have a proper understanding of the assignment flow at first and eventually redesigned it after spending significant time rereading.

At first, I was wasting a lot of memory and using regexes for every file, which ended up being overkilled. C's standard file functions are more than enough for parsing the strings I needed. Secondly, I changed from using string vectors to track errors to linked lists for multiple reasons. Using linked lists uses slightly more memory (storing duplicate HatterSpeak structs rather than simple strings) but it significantly simplifies the program design by not needing an entirely new ADT (a vector) or over allocating a massive buffer to avoid having to resize a vector. Linked lists, while duplicating data, are far more straightforward and efficient.