CS 261 Assignment 5

Sunday, May 31, 2020 7:37 PM

John-Francis Caccamo

Question 1:

Words that are anagrams (words that have the same letters but different arrangement) and have all upper-case or lower-case letters such as elbow and below or ELBOW and BELOW would both hash to the same key and collide in the same hash map with hash_function_1. This is due to the fact that hash_function_1 is simply converting each letter to its ASCII decimal value via the ord() function and calculating its sum in order to create a hash value that will be modulo'd by the capacity of the hash map in order to create the key.

Take for example elbow and hash_function_1()

Hash = ord(e) = (101) + ord(l) = (108) + ord(b) = (98) + ord(o) = (111) + ord(w) = (119) =

101 + 108 + 98 + 111 + 119 = 537

Now, below and hash_function_1()

Hash = ord(b) = (98) + ord(e) = (101) + ord(l) = (108) + ord(o) = (111) + ord(w) = (119) =

98 + 101+ 108 + 111 + 119 = 537

Via the commutative property of addition, these words hash to the same value.

Question 2:

hash_function_2's usage of the index variable to multiply to each letters' ASCII conversion while being incremented with each successive letter, introduces another variable that introduces another level of complexity that facilitates anagrams to have different keys and thus the ability to hash to different buckets. hash_function_1 had no such capability and simply hashed anagrams to the same key. Words with the same letters but in different order are hashed completely differently as opposed to hash_function_1.

Take for example elbow and hash_function_2()

Hash = ord(e) = (101 * (0 + 1)) + ord(l) = (108 * (1 + 1)) + ord(b) = (98 * (2 + 1)) + ord(o) = (111 * (3 + 1)) + ord(w) = (119 * (4 + 1))

101 + 216 + 294 + 444 + 595 = 1,650

Now, below and hash_function_1()

Hash = ord(b) = (98 * (0 + 1)) + ord(e) = (101 * (1 + 1)) + ord(l) = (108 * (2 + 1)) + ord(o) = (111 * (3 + 1)) + ord(w) = (119 * (4 + 1)) =

98 + 202+ 324 + 444 + 595 = 1,663

As demonstrated, hash_function_2 reduces collisions of anagrams.

Question 3:

Since hash_function_1 and hash_function_2 hash input keys differently, YES it is possible for empty_buckets() to return different values. For example, if all input keys had an anagram analog, there would be more collisions in the hash map using hash_function_1. More collisions results in unequal distribution among the buckets, meaning there would be less empty buckets with hash_function_1 (but more inefficiency with reading of the inevitable 2+ value linked lists, compromising the runtime).

However, the hash maps associated with both functions are getting the same input and thus will have the same size as each key-value pair increases hash map's member variable size by 1. Since the size will be the same for both hash maps, NO, table_load() will be the same for both hash maps and WILL return the same value.

Question 4:

The hash function is doing the bulwark of creating the randomization of the keys rather than the size of the hash map in this scenario. If we were dealing with keys with sequential numerical data (for instance keys = 1,2,3…, 10,000) and an input size of 2,000+ and simply creating keys via modulo'ing the input key by capacity, then having a hash map of capacity 997 would be evenly and properly hashing keys with multiples of 1,000. This is due to the fact that 997 is a prime number and has no factors but 1 and itself- the 2,000th input would be hashed to the 6rd bucket, the 3,000 would be hashed to the 9th bucket and so on (2,000 % 997 = 6, 3,000 % 997 = 9, etc.) . Otherwise a hash map of capacity 1,000 would be hashing all multiples of 1,000 to the same key since 1,000 is a common factor. This is where a prime bucket size would be preferential with its even distribution.

However, the above rationalization is mostly moot since if we were to have a input size of 2,000, 3,000 ... 10,000, etc., we would want to resize the hash map to reflect the size of the input to maintain O(1) constant look-up value time. If we have a hash map of size 997 or 1,000 then our input size should be around 1,000.

If we have an impartial and completely evenly distributing hash function (which is a quality every hash function should achieve) for an input size of approximately 1,000, then reducing our buckets from 1,000 to 997 simply results in 3 less buckets and thus less buckets for values to hash to and thus less empty buckets (and perhaps increasing potential for more collisions).

In addition, with a poor, suboptimal, and/or collision-causing hash function (or a circumstance where there is no hash function, at al), modulo'ing the key by a prime hash map size / bucket count, such as 997, will create a more even distribution of keys and thus less empty buckets across the hash map since 997 has no other factors by 1 and 997 compared to a hash map with 1,000 as its size since it has many factors (1,000 is base 2 and base 10).

So in either circumstance of having a good or bad hash function, reducing the number of buckets in the hash map from 1,000 to 997 does reduce the number of empty buckets.