CS261P- DATA STRUCTURES HW02

Name: Sameer Ganesh Shinde
UCI ID: 19644693 UCI NetID: sgshinde

1. Suppose that r = p/q, where p and q are integers, and that $0 < r < 1/k$ for some integer $k \geq 2$. Suppose we are maintaining a set of keys in a hash table, using linear probing. Keys may be added but are never deleted. Initially, the table size is q and there are p keys in the table (so, the load factor is initially r). As we add each key:
   - If the load factor will remain less than $k \cdot r$ after we add the key, we simply add it.
   - If adding the key would make the load factor be $k \cdot r$ or more, we allocate a new table k times as large as the original table (so that after the new key is added, the load factor is r).

   Show that the amortized cost per add operation is O(1).
   Assume:
   - the cost of adding an item, not counting the cost of re-allocating the table, is O(1).
   - the cost of re-allocating the table is proportional to size of the new table.

   Suggestion: Choose your potential function carefully!!

   Note: The potential function is figured out with the help of discussion with fellow classmate Nihal Gandhi

Ans:

Potential function:

$$\Phi = \frac{k}{k-1}(p - rq)$$

Key Information and assumptions:

Given:

k >= 2
p = number of keys in hash table
q = size of hash table
r = p/q

Assumption (Derived):

$0 < r < 1/k$

r <= load factor < k.r

Add an element in hash table: If the load factor will remain less than $k \cdot r$ after we add the key

O(1+c) actual time (Time to push an element plus constant additional time).

$\Delta\Phi = \Phi(\text{new}) - \Phi(\text{old})$

$$\Phi(\text{old}) = \frac{k}{k-1}(p - rq)$$

$$\Phi(\text{new}) = \frac{k}{k-1}(p + 1 - rq)$$

$\Delta\Phi = \frac{k}{k-1}(p + 1 - rq) - \frac{k}{k-1}(p - rq)$

$= \frac{k}{k-1}$

Amortized time = Actual time + $c \cdot \Delta\Phi$

$\leq 1 + c + \frac{k}{k-1}$

$= O(1)$

Add next element in hash table: If adding the key would make the load factor be $k \cdot r$ or more, we allocate a new table k times as large as the original table

O(p+1) actual time (Time to push an element plus constant additional time).

$\Delta\Phi = \Phi(\text{new}) - \Phi(\text{old})$

$\Phi(\text{old}) = \frac{k}{k-1}(p - rq)$

$\Phi(\text{new}) = \frac{k}{k-1}(p + 1 - rkq)$

$\Delta\Phi = \frac{k}{k-1}(p + 1 - rqk) - \frac{k}{k-1}(p - rq)$

$= \frac{k}{k-1}p + \frac{k}{k-1} - rqk.\frac{k}{k-1} - \frac{k}{k-1}p + \frac{k}{k-1}rq$

$= \frac{k}{k-1} - rqk.\frac{k}{k-1} + \frac{k}{k-1}rq$

From derivation, we know that

load factor = $\frac{p}{q}$

in this case load factor >= k r

Therefore, if we consider,

Load factor = kr

r= load factor / k

$= \frac{p}{qk}$

Let's substitute r with this value

$= \frac{k}{k-1} - \frac{p}{qk}qk.\frac{k}{k-1} + \frac{k}{k-1}\frac{p}{qk}q$

$= \frac{k}{k-1} - p\frac{k}{k-1} + \frac{p}{k-1}$

$= \frac{k}{k-1} - p(\frac{k-1}{k-1})$

$= -p$

Amortized time = Actual time + $c \cdot \Delta\Phi$

$\leq (p+1-p)\,c$

$= O(1)$

2. In class we described a way of preventing infinite loops in the set algorithm of cuckoo hashing, by counting steps and stopping when the count gets too large. This method has a problem: you need to know the constant factor to use in the Θ(log n) bound on the number of steps. Describe an alternative method of preventing infinite loops, without counting, by detecting a loop whenever it happens.
Ideally, your solution should

- Use a constant amount of additional memory
- Only perform a constant amount of additional work per step of the set algorithm
- Not use a step counter
- Detect any loop within a number of steps proportional to the number of keys involved in the loop
- Only report that there is a loop when the set algorithm really has an infinite loop

(Hint: consider what happens to the key given as the original argument to the set algorithm during a loop.)

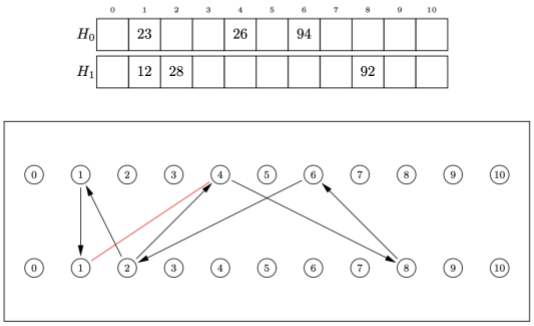
Answer:

In our classroom lectures, from the example of Visualizing Cuckoo hashing, we could not insert key 15, because there were 3 paths between 4U and 2L

4-5

Visualizing Cuckoo Hashing

In our example, we cannot add the key 15.
There is no way to orient the edges so that each vertex has only one incoming edge.
This is because there are three paths between 4U and 2L.



CompSci 261P—WQ 2019—©M. B. Dillencourt—University of California, Irvine

From this example, we can infer that when we are using 2 hash functions, at max there will be 4 positions to enter a key. That means if there are 3 paths between two elements, then there is a loop.
To address this, we can simply keep a counter against the given key. Obviously, we have to use a reference to this key for every hashing attempt. If we attempt to put the referenced key three times, then we are in a loop.

3. Suppose you have two Bloom filters FA and FB, each with the same number of cells and the same hash functions, representing the two sets A and B. Let FC = FA & FB be the Bloom filter formed by computing the bitwise Boolean and of FA and FB.

(a) FC may not always be the same as the Bloom filter that would be constructed by adding the elements of the set A ∩ B one at a time. Explain why not.
(b) Does FC correctly represent the set A ∩ B, in the sense that it always gives a positive answer for membership queries of all elements in this set? Explain why or why not.

Answer:

(a) Let's consider an example, if we add 6 to FA and 28,30 to FB then
H(6) = {1,3} H(28) = { 5,1} and H(30) = {3,8}
FA = {0,1,0,1,0,0,0,0,1,0,0}
FB = {0,1,0,1,0,1,0,0,1,0,0}
Then FC = {0,1,0,1,0,0,0,0,0,0,0}
Which represents 6. But we know that 6 is not a part of A ∩ B

To generalize we can say, if an element from one set, which is not present in other, maps to few cells, and same with other set, then the resulting FC will have some non-zero cells which may represent some other number which is not part of resulting set.

(b) Yes. Every element in A ∩ B will have its k bits set to non-zero from FA and FB. Therefore, resulting FC will also have that information by bitwise operation. Therefore, it will always correctly identify positive answer for the membership query of this set. But it can result in the cases of false positive as explained in the previous example.